National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

The University of Alberta

Design and Formal Specification of a Data Model and Language for a Database System for CAD Applications

by

Narayana Prasad Srirangapatna

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Doctor of Philosophy.

Department of Computing Science

Edmonton, Alberta
Spring, 1989

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-52860-5

Canada

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Narayana Prasad SRIRANGAPATNA

TITLE OF THESIS: Design and Formal Specification of a Data Model and Language for a Database System for CAD Applications

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Doctor of Philosophy

YEAR THIS DEGREE GRANTED: 1989

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) Narayana Prasad Srirangapatna

Permanent Address:

23, K.R. Vanam
Mysore - 570008
India

Dated 6 April 1989

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Design and Formal Specification of a Data Model and Language for a Database System for CAD Applications** submitted by **Narayana Prasad Srirangapatna** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**

.....
Supervisor

.....

.....

.....

Date

To my parents

ABSTRACT

The increasing importance of integrated databases in several specialized application areas like engineering design (CAD), interactive graphics, image processing, geographical data management, office automation et cetera, has pointed to the limitations of conventional DBMS and the need for more suitable database architectures. In particular, the special requirements of engineering design databases include: direct modeling of *design objects* and their *interrelationships* at different abstraction levels; *classification, aggregation, generalization* and *specialization*, multiple *representation, version* and *instance* abstractions; incremental and dynamic database schema definition and modification; complex, user-definable database *operations*, and semantic integrity and consistency *constraints* requiring the power of general programming languages for their specification and implementation; *concurrent access* and interactive *design transaction* management protocols; and flexible and high-level user interfaces. These requirements imply the need for "richer" semantic data models, complex data type systems, and object-oriented design paradigms. This thesis presents a *conceptual-level, semantic data model* that captures directly the notions of *entities, entity interrelationships* (representation, version and instance abstractions), and *structural abstraction hierarchies* (classification, aggregation, generalization and specialization). The model also provides a set of high-level *data model operations* that can be used to create and manipulate design objects. As well, the structural integrity constraints of *entity identity, entity typing*, and *referential integrity* are directly captured by the data model. More complex semantic integrity constraints (including *value domain, derived value, entity composition* and *interface*, and *mutual consistency* constraints) are also provided as explicit, user-specifiable constraints. The object paradigm provides a natural framework for the integration of structural and behavioural abstractions of the design database, and leads to the design of an *integrated database language* which combines a data type system that provides abstract data types

of *entity* (design object) and *value* (structured value), and structured types of *tuple, set* and *sequence* (complex objects) with a simple procedural, Pascal-like language. To date, conventional DBMSs have supported only rudimentary query and data manipulation languages and very simple structural integrity constraints. The proposed database language, on the other hand, enables user specification of complex, application-specific operations and semantic integrity and consistency constraints at the database schema level. These operations and constraints can be triggered by an integrity subsystem thus providing a means of checking and enforcing the integrity of the design database. These concepts and features of the design database system are illustrated with examples drawn from the domain of VLSI circuit design. Formal *algebraic* (axiomatic) and *model-based* (operational) specifications of a structured type system used in defining the semantic data model and the integrated database language are given. A subsystem that provides the structured values of sets, sequences, tuples and unions is also implemented. Finally, a formal denotational semantics specification of the semantic data model and the integrated database language is given using the VDM specification system. These specifications can be used as the basis for implementing particular subsystems of an actual design DBMS, or in rapid prototyping of an experimental version for design analysis, verification and refinement.

Acknowledgements

I am deeply grateful to my research advisor Professor William W. Armstrong for his invaluable guidance and constant encouragement throughout the course of my research study. This work owes a great deal to his persistent efforts to focus on basic and fundamental issues, and his insightful suggestions. I am also indebted to him for the financial support he provided to me during this period.

I am also highly thankful to the members of my examining committee, Professors Emil Girczyc, Mark Green, Tim Merrett and Tamer Ozsu for their careful reading and constructive comments which contributed significantly to refining the final version of this thesis.

I must also acknowledge the moral encouragement I received from my parents during the period of my study in a far away country. I am grateful to them for their unfailing confidence in me.

Last, but not least, the very helpful and cheerful spirit of all the administrative and technical support staff in the department made for a conducive environment in which to work. I am thankful to all of them.

Table of Contents

Chapter	Page
Chapter 1: INTRODUCTION	1
1.1. Computer-Aided Design Systems	1
1.2. CAD System Architecture	2
1.3. Advantages of a Design Database	6
1.4. Contributions of this Thesis	7
Chapter 2: CAD DATABASE SYSTEM ISSUES	10
2.1. Special Requirements of Design Databases	10
2.2. Limitations of Conventional Database Systems in CAD Environments	13
2.3. Elements of a CAD DBMS Architecture	18
2.3.1. A Conceptual-level, Semantic Data Model	20
2.3.2. A Mathematical and Logical Data Model	24
2.3.3. An Integrated Database Language: Integrating Data Types and Data Models	26
2.3.4. Application-specific Database Operations	28
2.3.5. Database Integrity Constraints	29
2.3.6. Concurrent Access Control and Transaction Management	32
2.4. Summary	33
Chapter 3: A SURVEY OF CURRENT RESEARCH IN CAD DATABASES	35
3.1. Introduction	35
3.2. Augmenting Classical Data Models	35
3.3. Semantic Data Models	37
3.4. Interfacing Data Models to Programming Languages	39
3.5. Integrated Database Languages	41
3.6. Object-oriented Database Management Systems	43
3.7. Comparative Analysis and Summary	44
Chapter 4: A SEMANTIC DATA MODEL FOR DESIGN DATABASES	48
4.1. Introduction	48
4.2. Entities	48
4.3. Entity Classes	49
4.4. Attributes	50
4.5. Constraints	52
4.5.1. Domain Constraint	52
4.5.2. Key Constraint	53
4.5.3. Class Membership Constraint	53
4.6. Entity Interrelationships	55
4.6.1. Representation Relationship	56

4.6.2. Version Relationship 57
4.6.3. Instance Relationship 58
4.6.4. Other Relationships 59
4.7. Schema Definition Constraints 63
4.7.1. Attribute Redefinition 64
4.7.2. Domain Constraint Redefinition 64
4.7.3. Key Redefinition 65
4.7.4. Relationship Definition 65
4.8. Attribute Value Table 67
4.9. Data Model Operations 67
4.10. Structural Integrity Constraints 76
4.11. Database State 77
4.12. Database Operations 78
4.12.1. Data Manipulation, Querying and Retrieval 81
4.12.2. Derived Attributes 82
4.12.3. Complex Semantic Integrity and Consistency Constraints 82
4.13. Partially Defined Design Objects in CAD Databases 84
4.14. Summary 87
Chapter 5: AN OBJECT-ORIENTED DATABASE LANGUAGE FOR A CAD DBMS 89
5.1. Introduction 89
5.2. Database Language Design Issues 90
5.3. Data Abstraction 91
5.4. Polymorphism and Property Inheritance 92
5.5. Data Persistence 95
5.6. Type System Implementation 98
5.7. System Architecture 103
5.8. Integrated Database Language for a D-DBMS: A Conceptual Basis 104
5.9. IDL System Design 110
5.9.1. Values 110
5.9.2. Type System 110
5.9.3. Variables 114
5.9.4. Expressions 115
5.9.5. Assignment 116
5.9.6. Procedures and Functions 119
5.9.7. Type Checking 121
5.9.8. Statements 122
5.9.9. Modularization 125
5.10. Summary 125
Chapter 6: DESIGN DATABASE SPECIFICATION: AN ILLUSTRATIVE EXAMPLE 127

6.1. Introduction 127
6.2. Database Scheme 127
6.3. An Utility Library 137
6.4. Defining a 4-bit Adder 140
6.5. Design Database Processing 149
6.5.1. Logic Simulation 150
6.5.2. Timing Analysis 155
6.6. Summary 159
Chapter 7: FORMAL SPECIFICATIONS 160
7.1. Introduction 160
7.2. Vienna Development Method 162
7.3. VDM Specifications 163
7.3.1. The VDM Type System 163
7.3.2. A VDM Model 173
7.4. Algebraic Specifications 175
7.4.1. Larch Auxiliary Specifications of the IDL Structured Types 175
7.5. SVIM: An Implementation of the Values and Operators of the IDL Structured Types 180
7.6. Denotational Semantics 184
7.7. A Denotational Semantics for IDL 187
7.7.1. Abstract Syntax 187
7.7.2. Semantic Domains 193
7.7.3. System State 194
7.7.4. Semantic Well-formedness and Meaning Functions 196
7.8. Summary 204
Chapter 8: CONCLUSION 205
8.1. Summary 205
8.2. Future Research 206
8.2.1. DBMS/Design Tool Interface 206
8.2.2. Database System Services 207
8.2.3. Other Data Models 209
References 212

List of Tables

Table	Page
3.1 CAD Database System Features: Comparative Analysis	47
4.1 3-valued Logic	52
4.2 Partial Database Schema	72
4.3 Partial Database Schema	73
4.4 Partial Database Schema	79
4.5 Partial Database Schema	81
5.1 IDL Operator Precedence Table	115

List of Figures

Figure	Page
1.1 CAD System Architecture: File-based System	4
1.2 CAD System Architecture: Database-based System	5
2.1 A Partial Composition Hierarchy of a VLSI Chip	11
2.2 A Layered Architecture for a CAD DBMS	25
2.3 Correspondence between Data Models and Data Types	27
4.1 Entity Class Composition (Aggregation) Hierarchy	62
4.2 Entity Class Generalization/Specialization Hierarchy	63
4.3 Cross Relationship Constraint	66
4.4 *Is-a* Hierarchy of Classes	72
4.5 *Is-a* Hierarchy of Classes	74
5.1 Variables and Values in IDL	118
5.2 Variables and Values in IDL: The State after specified Assignments	119
6.1 *Has-comp* and *Is-a* Class Interrelationship Hierarchies	134
6.2 4-bit Adder: VLSI Circuit Schematic	142
6.3 Full Adder: VLSI Circuit Schematic	143
6.4 Half Adder: VLSI Circuit Schematic	144
6.5 4-bit Adder: *Has-comp* and *Is-a* Class Interrelationship Hierarchy	145
6.6 Logic Simulation: Data Structures and Algorithms	152
6.7 Timing Analysis: Data Structures and Algorithms	156

Chapter 1

INTRODUCTION

1.1. Computer-Aided Design Systems

Engineering design is a complex process, difficult to describe formally. Design approaches and methodologies vary greatly depending on the application domain (mechanical engineering, VLSI, architectural design, chemical process engineering etc.), the specific design problem and the individual designer involved [HaL84, Seq83]. Broadly, however, engineering design can be described as a process that begins with a set of (possibly vague and incomplete) requirement specifications, involves a series of successive transformations (iterative improvements) of desired design parameters, and finally results in a set of specifications detailed enough for manufacturing or building the desired product, apparatus or structure. Typically, the design problem is very complex and has to be recursively "decomposed" into smaller and simpler problems each of which is solved separately, and the individual solutions are finally combined. *Computer-Aided Design* (CAD) is a design process in which a variety of computer-based tools and techniques are employed to aid in the design process. The increasing complexity, sophistication and scope of CAD applications has highlighted the need for and the importance of an integrated *design database* to store and manipulate large amounts of design data. This has also motivated the development of *integrated design environments* [AGS84, KSS83], analogous to the concept of programming environments for software engineering. The design environment provides a set of software tools, languages and services for interactive design activities. These include database management systems (DBMS), friendly user interfaces, messaging and communication facilities, and a host of software engineering tools like editors, compilers, interpreters, query and command languages etc. The design environment also provides an *integrated interface* which allows data definition,

manipulation, querying, computation and communication to be done without having to switch between different "modes" at the user's level; as well, it provides a *modular software architecture* which allows easy and incremental additions and modifications of the design system components. Such an interactive environment enables the design activity to be carried out iteratively at several abstraction levels ranging from functional specification at the top level to detailed physical specifications at the bottom level. Examples of such integrated design environments include Bell Labs' Designer Workbench [O'79] and vdd (VLSI design database system) [Chu83], HP's CAEE (Computer Aided Electrical Engineering) system [LeO83], Carnegie Mellon University's SPICE (Scientific Personal Integrated Computing Environment) [Bar84], and NASA's IPAD (Integrated Programs for Aerospace-Vehicle Design) [Joh80].

1.2. CAD System Architecture

Traditional *file-based systems* are based on a sequential design flow paradigm in which each design tool reads design data from a design file (which may have been produced by another design tool earlier in the design process), processes the data, and writes another design file (to be processed by other design tools in the next stage of the design process) (Figure (1.1)). Such systems are characterized by a multiplicity of data formats, and a flow of design data through a sequence of design steps. In a VLSI circuit design system, for instance, the design tools consist of *graphic editors, design rule checkers, PLA generators, circuit extractors, electrical simulators,* and *layout plotters*. The design file encoding formats may be CIF ([MeC80]) for layout, CAESAR ([Ost81]) for circuit schematic, and ISP ([BeN71]) for functional specifications. Architecturally, such systems may be organized around the facilities provided by sophisticated *operating* and *file systems* (e.g. SCALD [McW78], Designer's Workbench (DWB) [O'79], Berkeley CAD Project [New81] etc.) or a customized *data manager* that provides a uniform interface for

access to all data (e.g. University of California, Berkeley's SQUID [Kel84], HAWK [Kel84], IBM's Engineering Design System [San82] etc.). The drawbacks of such file-based systems are: (1) it is difficult to maintain consistency of the design data in different files and in different formats when these can be modified independently; (2) it is very difficult, if not impossible, to propagate changes backward along the design sequence; and (2) the system is batch-oriented, and incremental and interactive processing is not feasible.

A *database-centred* CAD system architecture (Figure (1.2)), on the other hand, overcomes some of these drawbacks by means of a *Design Database Management System* (D-DBMS) which maintains all design data in a (logically) centralized database and provides concurrent access control, transaction management, stable storage and failure recovery services. A *Design System Interface* (DSI) layered on top of the Design DBMS can then provide access protocols, specialized subviews of the design data required by different design tools or interactive users, and feedback in case of design errors or inconsistencies. The advantages of such an architecture are described in the next section.

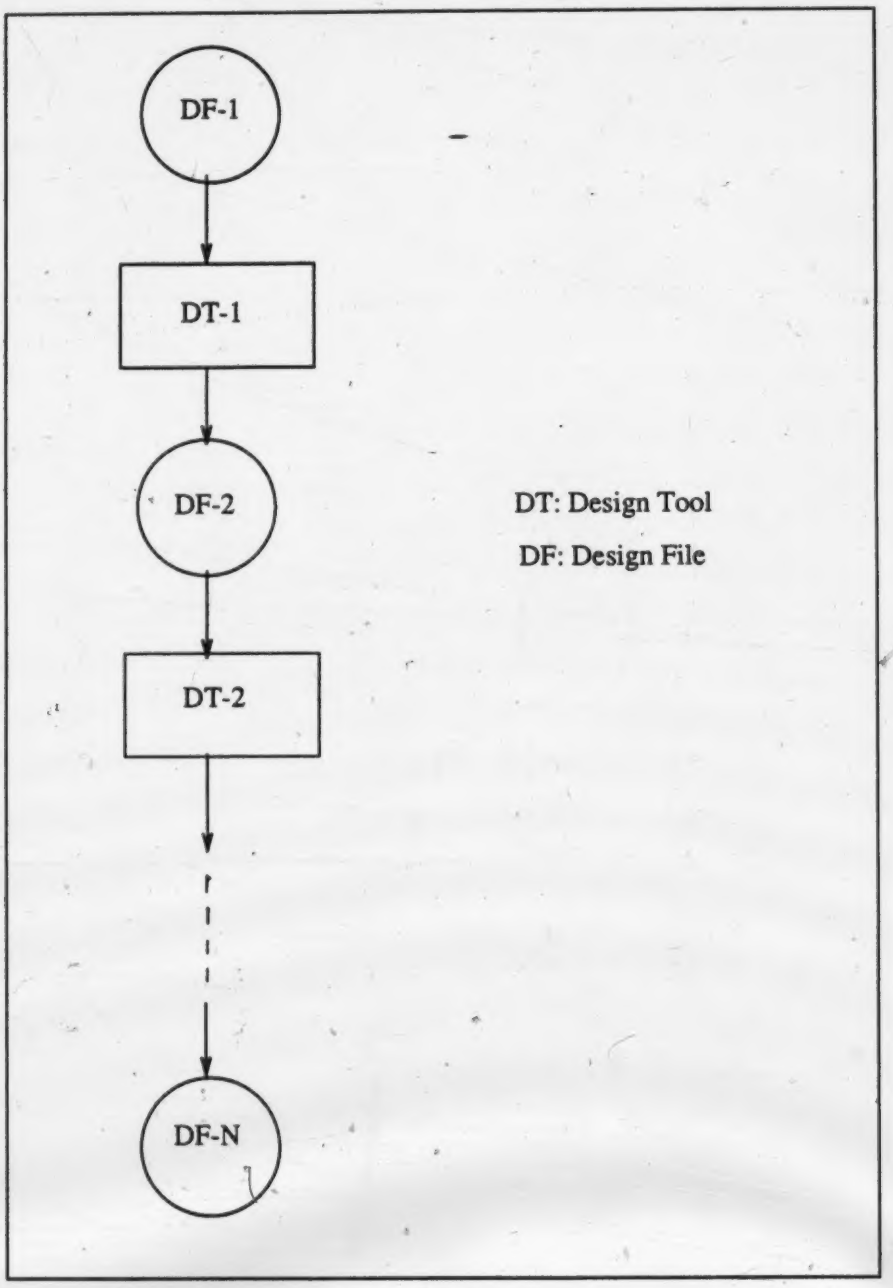

Figure 1.1 CAD System Architecture: File-based System

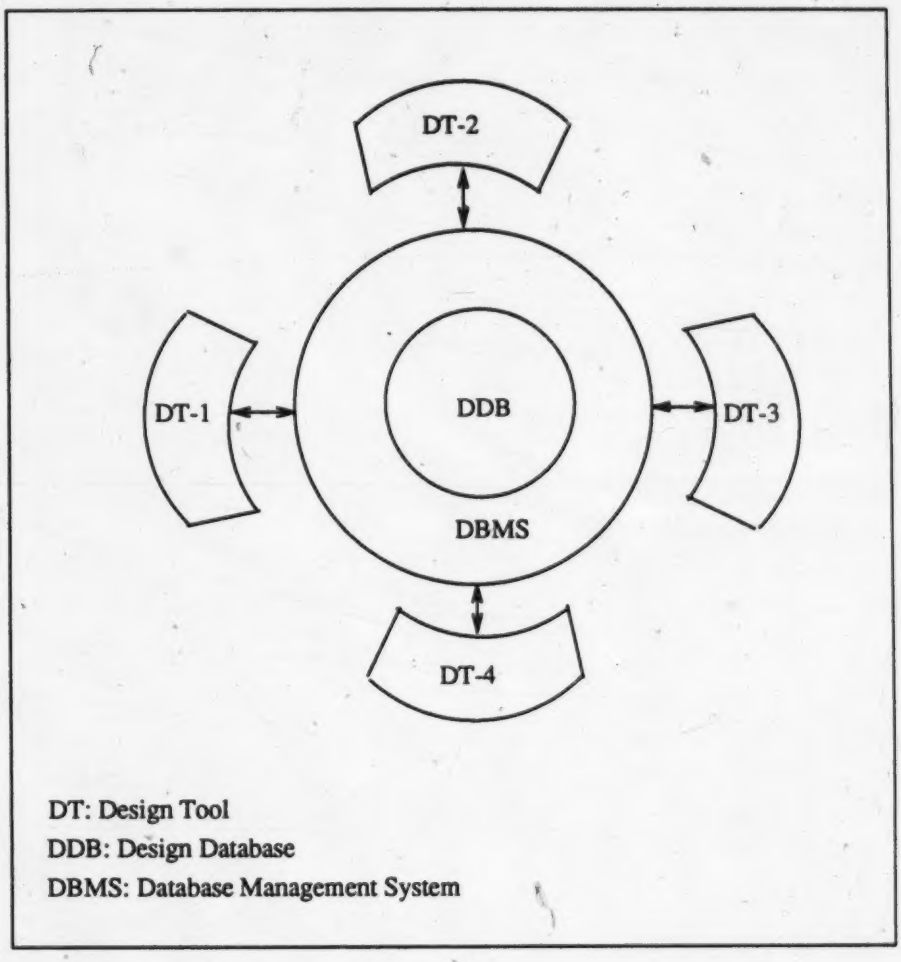

Figure 1.2 CAD System Architecture: Database-based System

1.3. Advantages of a Design Database

The principal advantages of using a centralized *design database* in a CAD system are:

• *Automation of data translation between different formats, views and representations required by different design tools*

CAD applications have evolved into systems consisting of a number of *design tools* each of which reads design data stored in a simple data file in a fixed format, processes the data in a batch mode, and generates output data in a fixed format, again stored in a simple data file. At each step of this multistage, batch-oriented design process, data translators and formatters are needed to transform and format data produced by one design tool into a form that is required by the next design tool in the design sequence. This transformation and formatting process can be more effectively carried out if all the design data is organized in a *design database* managed by a D-DBMS.

• *Centralization of code required to build and maintain efficient data storage, update and access structures*

This was one of the major benefits that motivated the development of DBMSs for large scale data processing applications where a large number of application programs operate on a common data base. This feature also provides *physical data independence:* data storage and access structures can now be changed without affecting the logical view of data encoded in the application programs.

• *Easier management of semantic integrity and mutual consistency among multiple representations of design data*

With the logical centralization of all design data under the control of a D-DBMS, it becomes possible to maintain consistency and correctness by specifying and enforcing database integrity constraints. This aspect is particularly crucial in the CAD environment

which is characterized by complex semantic integrity and consistency requirements.

• *Elimination of duplication and redundancy; data sharability; logical data independence*

Further, with the centralization of design data, duplication and redundancy of data is reduced and data sharability is increased greatly. Also, the D-DBMS provides, by means of a *schema* (internal) and *view* (external) definition facilities, *logical data independence*: each design tool can (within limits) define its own "logical view" of the same underlying database.

• *Modular system architecture*

The database-centred architecture also provides modularity, since communication between independently evolving design tools can be more easily established through the design database. Also, new design tools can be added, and existing ones modified without having a major impact on the rest of the system.

1.4. Contributions of this Thesis

The contributions of this thesis are as follows:

Chapter 2

(1) Characterizing design databases in terms of their special requirements which make them different from the conventional databases for business data processing.

(2) Showing how the conventional database systems do not meet these special requirements with respect to modeling expressiveness, complex operations, complex semantic integrity and consistency constraints, concurrent access control and transaction management protocols, and system interfaces.

(3) Defining an architectural framework for building a D-DBMS. The elements of this framework are (a) a conceptual-level, semantic data model; (b) an internal-level

logical data model; (c) an integrated database language; (d) concurrent access control and design transaction management protocols; and (e) complex operations and semantic integrity and consistency constraints. Only brief overviews of these elements are sketched here. Items (a) and (c) are discussed in more detail in Chapters 4 and 5.

Chapter 3

(4) A survey of current literature on database systems for non-traditional applications with particular reference to systems, models, techniques of specific relevance to design databases in a CAD environment.

Chapter 4

(5) Development and specification of a *conceptual-level, semantic data model* that captures the abstractions required in design databases and serves as the basis for an implementation of a D-DBMS for CAD applications.

Chapter 5

(6) Design and specification of an *integrated database language* that addresses the issues of data abstraction, polymorphism, data persistence etc. in the design database environment. The database language system provides an implementation of the conceptual-level, semantic data model.

Chapter 6

(7) Developing a methodology for design database specification and processing of a VLSI circuit design.

(8) Illustrating the methodology by application to the specification of a 4-bit adder circuit and its design processing tasks of logic simulation and timing analysis.

(9) Formal VDM (denotational) and Larch (algebraic) specifications of a type system that provides structured types and values. This type system serves as a building block in (a) the design of the semantic data model and the integrated database language; and (b) in their formal specification.

(10) A denotational semantics specification for the semantic data model and the integrated database language.

(11) The SVIM subsystem that implements the structured values specified in (7). This can be used as a basis for (a) a type system for the database language, and (b) a (VDM) denotational semantics specification interpreter which then defines an operational specification of the semantics as well as providing a rapid compiler prototyping or generating tool.

Chapter 2

CAD DATABASE SYSTEM ISSUES

2.1. Special Requirements of Design Databases

There are several unique and special requirements of a design database that are not met by conventional database systems. These requirements include:

• *Hierarchically composed entities and complex objects*

A CAD system manipulates hierarchically composed design entities with several different representations at different abstraction levels. For example, in a VLSI circuit design system, a *Chip* entity is composed of *Bus, ALU, Register Section,* and *Control Section* entities. The *Bus* entity may, in turn, be composed of *Adder, Zero Control, Shift Register, Shift Control,* and *XOR Control,* entities. The *Adder* may consist of a set of *Gate* entities and so on. A partial *composition hierarchy* of such a VLSI design database is shown in Figure (2.1). The design object composition hierarchy can, in general, be defined recursively: an entity belonging to a specified entity class can contain other entities, including entities in the same class, as components; however, an entity cannot contain itself as a component. Thus a *Cell* entity can contain other *Cell* entities as components but not itself as a component. Hence, the VLSI circuit cell composition hierarchy is an acyclic directed graph with nodes representing entities and edges representing the *Has-component* relationship. *Complex objects* are structured collections of entities or basic data type values which are to be manipulated as "atomic" objects. In the VLSI design database for example, it is necessary to manipulate a set of *Gate* entities or a sequence of *Point* entities as atomic objects.

• *Multiple representations for entities*

Design databases are also characterized by multiple representations for entities

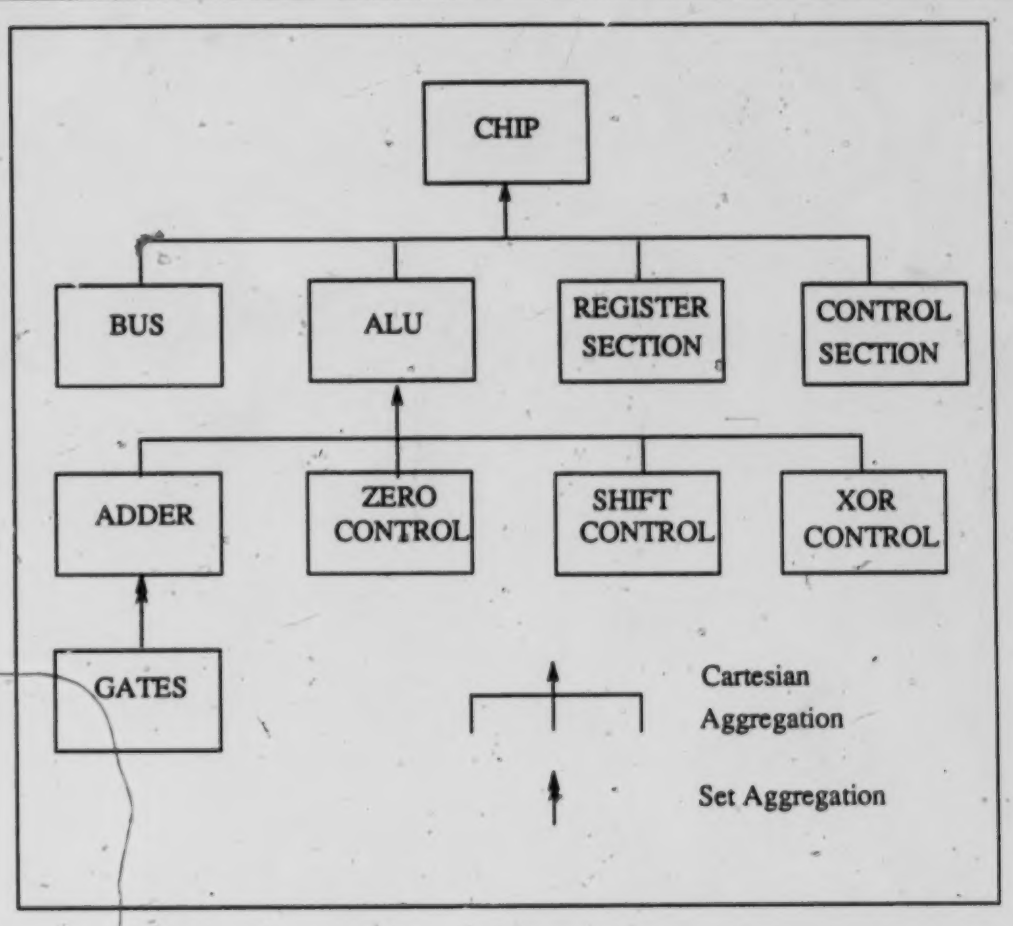

Figure 2.1 A Partial Composition Hierarchy of a VLSI Chip

corresponding to different abstraction levels at which these entities can be viewed. For example, a VLSI chip may be represented as a *layout geometry*, a *transistor network*, a *logic circuit* or a *functional block schematic*. The D-DBMS should set up these multiple representations and track their mutual consistency through updates and changes to the database made by the designer.

• *Multiple Entity Interrelationships*

Design databases are also characterized by multiple, "typed" entity interrelationships, each with its own application-specific, user-defined semantics. Common entity interrelationships include *Version-of* and *Instance-of* relationships between entities. The *Version-of* relationship relates multiple *versions* which are different implementations with the same functionality or interface, or different implementations obtained by updating specified property values. For example, there may be several versions of an ALU entity with the same functionality but based on different device technologies. The *Instance-of* relationship relates multiple instantiations of a standard or library part. Each instantiation (or instance) has its own properties (corresponding to the instantiation parameters) in addition to those of the standard or library part. For example, a standard cell can be instantiated with appropriate translation, rotation and scaling coordinate transformations, and used as a component entity in defining a larger cell or an entire chip. The associated relationship semantics may take the form of operations that can be applied to the relationship instances, integrity and consistency constraints on entities participating in the relationship, their association cardinalities, and/or protocols (that trigger constraint-checking or semantics-preserving operations when specific relationship instances are created or deleted from the database).

• *Complex Semantic Integrity and Consistency Constraints*

Design databases are also characterized by complex semantic integrity and consistency constraints on design entities and their interrelationships. These constraints may require the full power of partial recursive functions for their specification. This complexity arises from the complex structure of entity composition hierarchies, the need for complex objects, the multiple representations at different abstraction levels, and the variety of entity interrelationships. For example, in the VLSI circuit layout database there could be a semantic integrity constraint that boxes of a cell located on the same layer do not over-

lap. If the cell entities are recursively composed from other cell instances, then a recursive function is required to specify and check the above constraint. This is one example of the class of the so called "transitive closure" computation problems [Cle81].

• *Long-duration interactive design transactions*

The design process is, by nature, iterative, incremental and interactive. Design transactions can, therefore, last for long time periods. Consequently, conventional database notions of concurrent access control, transaction correctness and atomicity have to be redefined if we are not to lock out large parts of the design database, or make it effectively a single-designer system. There is thus a need for new protocols for concurrency control and transaction management.

There are other important requirements for design databases including incremental schema definition and modification, fast response, flexible interfaces and modular system architectures that we have not discussed in detail here, but which are important for a practical design database. All of these requirements make design databases different from conventional databases used in business data processing applications.

2.2. Limitations of Conventional Database Systems in CAD Environments

In this section, we consider the limitations of conventional database systems based on the classical data models which have proved successful in business data processing applications. Examples of such systems include INGRES [Sto74], DB2/SQL [Cha76], RDB etc. The fundamental limitations of such systems in a CAD application environment are:

• *Modeling Expressivenes*

All the general-purpose database systems available to date are based on one of the three classical data models: hierarchical, network and relational. There are two aspects of

modeling expressiveness that limit the applicability and ease of use of these conventional DBMSs in CAD applications. These are:

(1) *Entity:* The classical data models have no notion of an *entity* as a uniquely identifiable object in the application domain. Entities are represented as records (network), segments (hierarchical) or tuples (relational), and *entity identity* is mapped onto physical or logical addresses, or "key" attribute values. This leads to a proliferation of explicit integrity constraints; for example, the key uniqueness, subset or foreign key constraints in the relational model. Also, loss of entity identity makes it impossible to define operations which take entities or complex objects as arguments, and return them as results.

(2) *Abstraction Mechanisms:* The classical data models are "representation-oriented" and do not provide any abstraction mechanisms like *classification, aggregation, generalization* and *specialization* (formally defined in Section 2.3.1). These abstraction mechanisms are particularly useful in the design environment to deal with large, complex structures and relationships by abstracting essential properties. For example, by grouping together a set of entities which share common properties into an *entity class* (classification) a user can define class-specific (or type-specific) operations on these entities (*entity typing*). Aggregation abstraction is essential for dealing with complex objects; and a *superclass/subclass* entity class interrelationship (generalization/specialization) provides a means of viewing entities at different abstraction levels. For example, the classical relational model is "flat" in that its primitives (tuples and relations) cannot be nested to define hierarchically composed entities or grouped together to define complex objects. The connections among the components of a hierarchically structured entity or a complex object have to be defined in terms of specified property (attribute) values. This makes access and manipulation of complex objects very difficult, contrived and unnatural by requiring the user to specify explicitly the connections defining the hierarchically struc-

tured entity or the complex object. None of these abstraction mechanisms is provided by the classical data models. Also, the relational model eliminates the distinction between entities and relationships by implementing both as tuples - *semantic relativism* - which makes it impossible for the system to automatically enforce associated semantic constraints like referential integrity. For example, semantic relativism makes it impossible for the DBMS to enforce a semantic constraint like *referential integrity* (which constrains all entity-valued references to refer to entities which exist in the database). Thus, if a tuple is deleted from a relation, the DBMS cannot ensure that no tuple in any of the other relations contains a reference to the deleted tuple (via an attribute value). This limitation can, of course, be remedied by means of "foreign key attributes", "domain subset constraints" or "abstract domains" as in Codd's RM/T [Cod81]. The system can now ensure referential integrity by checking that each referencing attribute value is a valid reference.

• *Complex, user-defined, database operations:*

The conventional database systems provide only a set of primitive, low-level data manipulation, navigation and retrieval operations. Even the "high-level" relational algebra or relational calculus operations in relational database systems are "structure-oriented" and "type-independent." Furthermore, most high-level query languages (SQL [Cha76]) do not support recursive queries or general-purpose computation (*Turing computability*). Any sophisticated data manipulation or computation in such systems requires an embedding of a data access sublanguage in a general-purpose host programming language language (EQUEL [Sto74]. This has serious problems of (i) "impedance mismatch" between the abstractions supported by the database system and those supported by the host programming language; (ii) lack of data persistence; (iii) lack of data polymorphism etc. We shall discuss these issues in greater detail in Chapter 5. It is, therefore, important that the design database system provide an integrated *database language* through which users can

define sophisticated, application-specific operations on data, and *derived* (or computed) *properties*. For example, an operation *Scale* can be defined to geometrically scale a cell entity stored in a VLSI circuit design database.

• *Complex Semantic Integrity and Consistency Constraints*

The design database is characterized by a number of potentially complex semantic integrity and consistency constraints arising from the complex composition structure of entities, their interrelationships and multiple representations at different abstraction levels. These may be *value domain constraints* ("fan-in" and "fan-out" constraints on a VLSI circuit gate entity), *composition and interface constraints* (boxes on the same layer of a cell layout do not overlap or cell interconnects terminate properly etc.), and mutual *consistency constraints* (logic, circuit, transistor network and geometric layout representations of a chip are consistent). These explicit, user-defined constraints may be specified as assertions of first-order logic, or as partial recursive functions. For example, the composition constraint that for any cell layout boxes on the same layer do not overlap is specified by a recursive procedure that, given a cell, descends a recursively-defined cell hierarchy to check the constraint. Again, conventional database systems do not handle such complex semantic integrity constraints. The classical data models provide only simple "structural integrity" constraints (e.g. one-to-many association cardinality from owner to member records in a CODASYL/DBTG set; key uniqueness in a relation etc.). This implies, as with operations, that complex constraints can only be checked and enforced by application programs with embedded data access statements in them, thus leading to complexity at the user and application levels, and consequent redundancies and inconsistencies.

• *Concurrent Access Control and Design Transaction Management*

Here again, the conventional database systems which rely on classical concurrent access

control and transaction management protocols are inadequate. Any concurrent access control scheme for design databases has to deal with the following issues:

(1) What are the concurrent access and update semantics with respect to complex objects and entity composition hierarchies? This issue interacts strongly with how complex objects and entity composition hierarchies are implemented, and what are the semantics associated with their creation, deletion and manipulation.

(2) Given the above considerations, what kind of concurrent access protocols are required? If locking techniques are to be employed, what is the granularity of locking?

(3) What are the semantics of a design transaction, or equivalently, how are design transactions defined in terms of the integrity and consistency constraints they are to preserve? How are long-duration, interactive design transactions to be handled?

Concurrent accesses in conventional database systems involve very simple "read-modify-write" sequences, each of which affects a small data unit (a record or a tuple) and takes a very short time (a few seconds). Furthermore, the integrity contraints with respect to which the transactions are defined are simple enough to be checked "on-the-fly". The classical techniques of 2-phase locking or time stamping are defined with respect to these characteristics and, therefore, are quite inadequate in dealing with fundamentally different modes of concurrent access and complex constraints associated with transcations in a design system.

• *System Interfaces*

Design systems consist of several design tools or application programs which operate on the data stored in the design database. It should be possible to quickly and easily interface these tools to the database system. The system should also provide an interactive user interface so that the designer can inspect, query and modify the design database

interactively. A user interface to the design database can be defined in terms of one or more of the following: (1) a set of *database routines* that can be linked to the design tool at load time; (2) an *interprocess communication protocol* that can provide access from an independently executing application process through the D-DBMS process to the design database; and (3) an *integrated database language* that provides both interactive and programmed access to the database. Conventional systems provide only the first type of interface, or equivalently, one which involves preprocessing embedded database statements into procedure/function calls to the DBMS, which can then be processed as in (1) above. An integrated database language has to be a general-purpose, procedural language which has built into it the data model abstractions of the design database so as to make it not only possible, but also very natural and straightforward, for the user to specify complex database operations and semantic integrity and consistency constraints.

2.3. Elements of a CAD DBMS Architecture

From the discussion of the requirements of design databases and the limitations of conventional database systems in addressing these requirements, we are led to specify a database system architecture that consists of the following major elements:

(1) A *conceptual-level, semantic data model* that models directly both the structural (objects) and behavioural (operations) abstractions of the design application. Thus, constructs of this data model specify entities, entity composition hierarchies, complex objects, entity classes, representation, version, instance interrelationships between entities, and so on.

(2) A *mathematical and logical data model* which provides data structures and operations suitable for implementing the conceptual-level, semantic data model abstractions. That is, entities, entity classes, composition hierarchies etc. of the semantic

data model (the *conceptual-level*) can be implemented in terms of the pure relational model constructs (tuples, relations, foreign keys etc.), or in terms of abstract and structured types of VDM (set, sequence, map types etc.) (the *logical level*). Also required is a logic to formulate assertions about these structures and operations and to prove theorems about properties of models and implementations.

(3) An integrated, formally defined *database language* that not only supports the data model abstractions but also combines a facility for general-purpose computation with features like data persistence, polymorphism, etc. required in a database environment. Such a database language can be used to specify both complex, application-specific *operations* on data, and complex, semantic integrity and consistency *constraints*. Further, the database language should provide a single, unified interface for interactive data definition (commands or statements), and programmed data access and manipulation (database function or procedure calls).

(4) *Operational protocols* that regulate concurrent accesses to the database and implement interactive design transactions by enforcing specified semantic integrity and consistency constraints. These protocols take the form of complex, user-defined operations (specified as procedures or functions written in the integrated database language) that are triggered when the user invokes the data model operations that manipulate entities or entity interrelationships stored in the database.

(4) A *modular system architecture* which provides convenient interfaces to other CAD system components, including design tools and system utilities, and enables easy addition of other functional modules.

2.3.1. A Conceptual-level, Semantic Data Model

In this section we describe the design database abstractions captured by a conceptual-level, semantic data model. The conceptual view of a design database consists of *design entities* which are typically composed of other design entities (*entity composition hierarchy*); these entities have properties (*attributes, relationships,* and *constraints*); entities have multiple *representations* and *versions* existing simultaneously. The design database semantics are partly defined by a set of integrity constraints including *structural integrity constraints* (unique entity identities, and referential integrity), and complex *semantic integrity and consistency constraints* (value domain, value uniqueness, functional dependency, entity composition and equivalency constraints). Design entities may exist in partially or fully defined "states" in the database. We define the following abstractions to model and structure this conceptual view of design databases:

- *Database Entities:* correspond to design objects in the design application domain.

  Definition 2.1: Every database entity has a unique system-defined *identifier* (also called *surrogate* as in Codd's RM/T [Cod81]).

- *Classification:* The *classification* abstraction is defined in terms of *entity classes* [HaM81].

  Definition 2.2: An *entity class intension* is the collection of properties (*attributes* and *constraints*) shared by all the entities belonging to the class. An *entity class domain* is the set of all entities that posses the properties defined by the corresponding entity class intension. An entity class domain can be empty or non-empty, finite or infinite, and it can be generated and/or recognized by some procedure. An *entity class extension* is the subset of the entity class domain that is actually stored in the database at a given time instant. The entity class intension (properties) is statically defined while the entity class extension is created and modified dynamically by

inserting into it newly created entities and deleting from it existing entities.

- *Aggregation:* The *aggregation* abstraction enables a collection of entities or data values to be viewed as an "atomic" object with its own entity identifier.

  Definition 2.3: The *Cartesian*, *set* and *sequence* aggregation abstractions define, respectively, tuple (t), set (s) and sequence (q) objects as follows:

$t = (a_1 = e_1, ..., a_N = e_N)$	/* $a_1, ..., a_N$ are attributes; $e_1, ..., e_N$ are entities or basic data type values */
$s = \{ e_1, ..., e_N \}$	/* $e_1, ..., e_N$ are entities or basic data type values */
$q = < e_1, ..., e_N >$	/* $e_1, ..., e_N$ are entities or basic data type values */

  The components $e_1, ..., e_N$ are entities which could potentially be members of multiple entity class extensions (and hence belong to multiple classes) or basic data type values. The "type" information associated with the components is then one of the elements from the set { "Entity", "Integer", "Real", "Boolean", "String" }. The type "Entity" is specified by listing an entity class.

  The tuple, set and sequence objects obtained by the aggregation abstraction can themselves be viewed as entities, and therefore, can be members of the extension of some entity class. For example, the tuple object t could belong to an entity class (say) *Tuple-object* with attributes (properties) $a_1, ..., a_N$ and corresponding types entity classes $E_1, ..., E_n$. The set and sequence abstractions can be used to model complex objects which are sets and sequences of entities or other basic data type values.

- *Generalization/Specialization:* The *generalization* abstraction enables entities in different entity classes to be grouped together in a *superclass* that abstracts (or factors out) their shared common properties.

  Definition 2.4: The generalization of entity classes $E_1, .., E_n$ defines an entity

superclass E such that:

$$\text{Ext}(E) = \text{Ext}(e_1) \cup ... \cup \text{Ext}(E_n)$$

where *Ext(E)* denotes the extension of class E.

The specialization abstraction (inverse of generalization) enables entities in an entity class to be partitioned into different *subclasses* according to the additional properties that distinguish entities in them.

Definition 2.5: The specializations of an entity class E define entity *subclasses* E_1, ..., E_n such that:

$$\text{Ext}(E_1) \cup ... \cup \text{Ext}(E_n) = \text{Ext}(E)$$
$$\text{Ext}(E_i) = \{ e \mid e \in \text{Ext}(E) \ \& \ e \text{ has properties defined by the intension of } E_i \ \}$$

These abstractions can also be defined in terms of the *Is-a (subclass of)* relation:

$$\text{Is-a} = \{ (E_1, E_2) \mid (E_1 \text{ is a subclass of } E_2 \text{ OR } E_2 \text{ is a superclass of } E_1) \text{ AND } E_1 \text{ inherits the properties of } E_2 \ \}$$

such that if $(E_1, E_2) \in Is\text{-}a$, then E_1 is a specialization of E_2. The *Is-a* relation thus defines a *property inheritance hierarchy* of entity classes.

- *Multiple Representations:* Typically, design entities are modeled at different levels of abstraction for the purposes of design, analysis and evaluation.

Definition 2.6: The multiple representation abstraction is modeled in terms of the *Repr-of* relation defined over the domains of a *representation* entity class E_1 and an *abstract* entity class E_2 as:

$$\text{Repr-of} \subseteq \text{dom}(E_1) \times \text{dom}(E_2)$$
$$(e_1, e_2) \in \text{Repr-of} \iff \text{entity } e_1 \text{ is a "representation" of entity } e_2$$

where *dom(E)* represents the domain of entity class E. The *Repr-of* relation enables the system to track the relationships among multiple entity representations to enforce associated semantic integrity and consistency constraints.

- *Multiple Versions:* A given design entity may have different versions intended to be functionally equivalent. These presumably have different internal structures or attributes but the same external interface. *Implementation versions* are a set of alternative implementations to a functionally specified entity. *Update versions* are a set of refinements obtained by successive updates to entities in a "version set".

  Definition 2.7: The multiple version abstraction is modeled in terms of the *Version-of* many-to-1 mapping defined from the domain of a *version* entity class E_1 to the domain of a *generic* entity class E_2 as:

  Version-of: $\text{dom}(E_1) \rightarrow \text{dom}(E_2)$
  Version-of$(e_1) = e_2$ <=> entity e_1 is a "version" of entity e_2

  Again the *version-of* mapping enables the system to track the relationships among multiple versions and enforce associated semantic integrity and consistency constraints.

- *Multiple Instances:* A design may consist of multiple "instances" of a given design entity each of which augments the design entity by defining additional properties; there may also be several copies of a design entity existing in the database at the same time. For example, a design entity from a "standard part library" may be used as a component several times in a given design.

  Definition 2.8: The multiple instance abstraction is modeled in terms of the *Instance-of* many-to-1 mapping defined from the domain of a *instance* entity class E_1 to a *master* entity class E_2 as:

  Instance-of: $\text{dom}(E_1) \rightarrow \text{dom}(E_2)$
  Instance-of$(e_1) = e_2$ <=> entity e_1 is an "instance" of entity e_2

  Again the *Instance-of* mapping enables the system to track the relationships among multiple instances and enforce associated semantic integrity and consistency constraints.

To enforce the semantics associated with the three entity interrealtionships, additional equivalency or other constraint-checking operations must be associated with each realtionship by the application programmer.

- *Data Model Operations*: are basic operations to create, delete, retrieve and otherwise manipulate entities, entity properties and interrelationships. These correspond to the basic relational algebra (or relationa calculus) operations of the relational data model. These are described in detail with reference to the conceptual-level, semantic data model defined in Chapter 4.

2.3.2. A Mathematical and Logical Data Model

This model provides a formalization of the data structures and operations of the subsystem that implements certain abstractions of the conceptual-level, semantic data model (like set and sequence values (complex objects)). As shown in Figure (2.2), this defines an implementation layer that interfaces the semantic data model layer with the file system layer. If this logical data model layer provides the appropriate data structures and operations, the specification and implementation of the semantic data model layer can be simplified to a great extent. Thus, for example, the implementation of entities, entity classes, and complex objects of the semantic data model can be defined in terms of a set of basic, parametrizable data structures and operations (data types) provided by the internal-level, logical data model. Given the complex abstractions to be supported by the semantic data model (Section 2.2), the relational model proves too "limited and low-level" for use as the internal-level logical data model because of its highly restricted data structures and operations. We therefore need to define a more suitable logical data model.

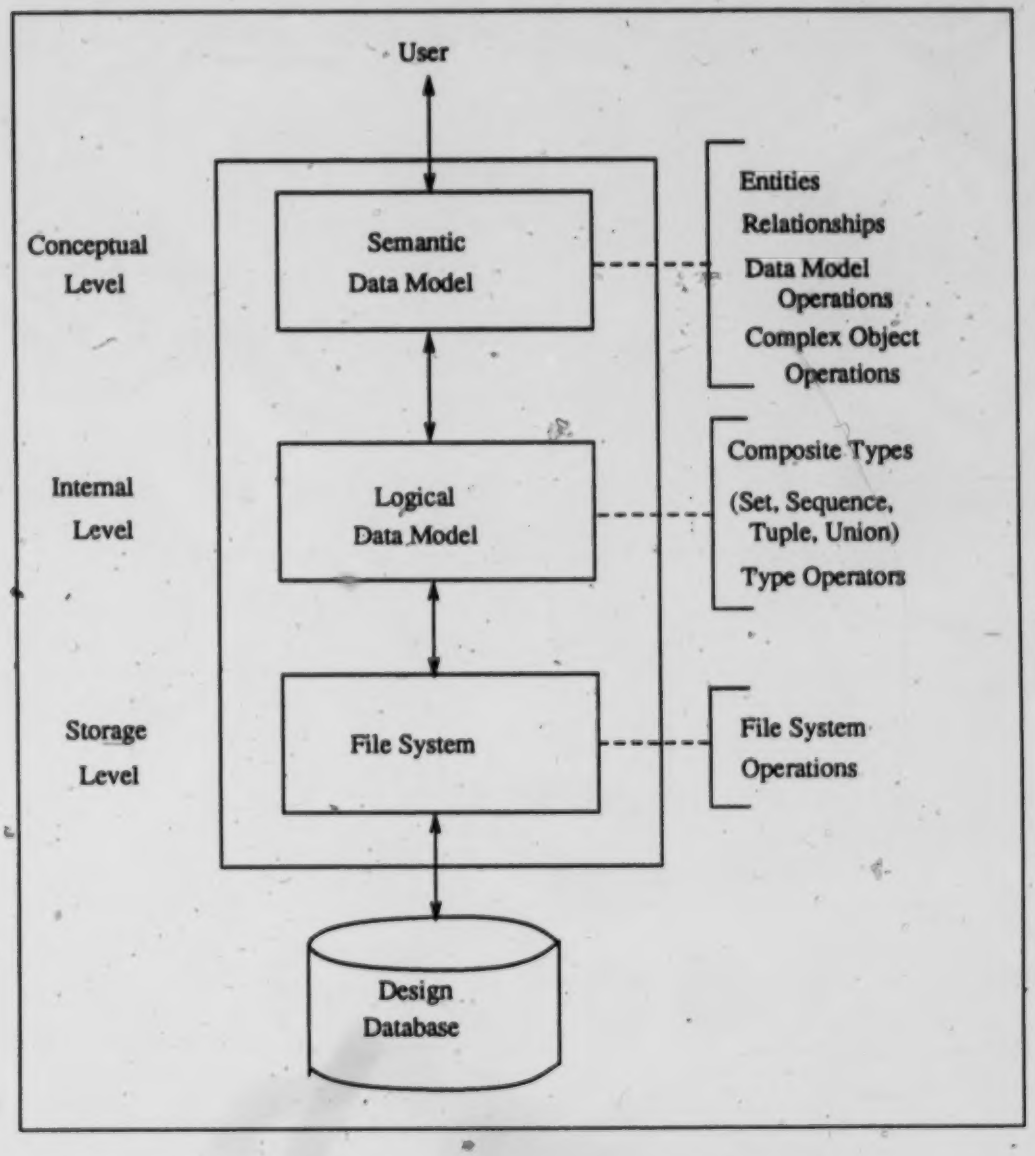

Figure 2.2 A Layered Architecture for a CAD DBMS

2.3.3. An Integrated Database Language: Integrating Data Types and Data Models

To date, database applications have been limited by the "impedance mismatch" between classical data models (with their data abstractions) and conventional high-level languages (with their data types). The traditional approach of embedding a data manipulation language in a host language has resulted in complicated, clumsy and ad-hoc interfaces defined in terms of "work areas," "currency indicators," et cetera [Bro80]. A more recent approach to integrating data models with programming languages seeks to extend the programming language with special data types that capture the data model abstractions. Examples of such systems include Aldat [Mer78], Pascal/R [ScM80] and Modula/R [Rei84] which add a *relation* data type to their host languages. Extending this approach we can now design an integrated *database langauge* that is equipped with a range of data types needed to capture all the complex data model abstractions. Database specifications written in such a language can now be statically checked for "consistency and correctness" by well known techniques of type inferencing and checking. This leads us to the idea of a hierarchical mapping between data models and data type systems [Bro80] as shown in Figure (2.3). A database instance is now a collection of values (of specified data types), the database schema is a set of data type specifications, and the data model is just the underlying type representation (data structure). The features of databases to be supported by such an integrated database language are:

- *Strong Typing* - to support (and enforce) semantically meaningful operations on database objects. Brodie's proposal [Bro80] of "interpreted types" (user-defined types whose domain is specified in terms of an underlying type (e.g. Integer) which make two types defined on the same underlying domain "incompatible") is an example of strong typing used to enforce application semantics.

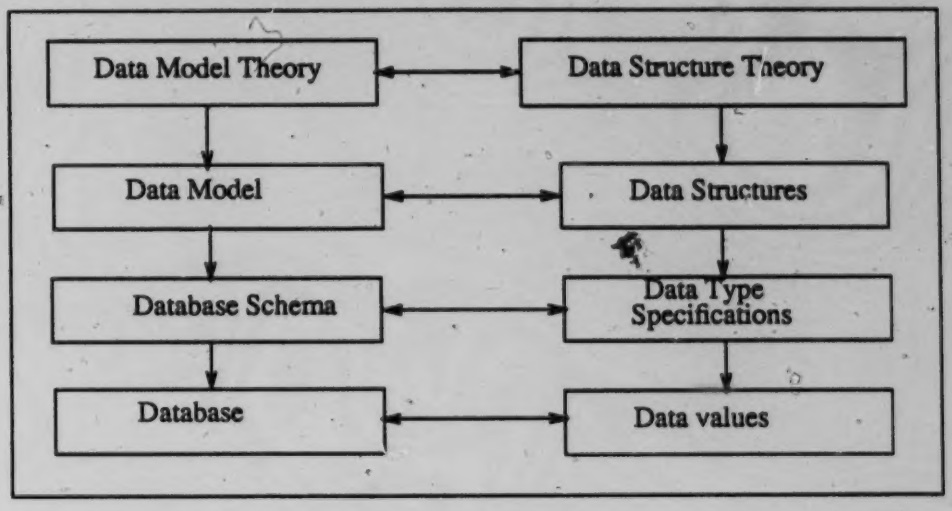

Figure 2.3 Correspondence between Data Models and Data Types

- *Data Abstraction* - the representation of the data type is hidden from the application and the only access to data values is provided through the type's operators.
- *Generic Operations and Polymorphism* - the type system should also be powerful enough to support specification of "generic operations" which can operate on values of more than one type, and "polymorphic values" which belong to more than one type [BuA86].
- *Data Persistence* - data values exist independently of an execution instance of the database program. This involves mapping the data type values into a set of "persistent data structures" which are replicated on secondary storage by the system. When the database is accessed subsequently through another program, the values should be recoverable along with the associated type information. Thus, the database language should ensure that while a value persists so does its type.

- *Data Sharing* - among multiple instances, applications, views and users. This involves notions of "program modules" and "scopes" [MSR84] and concurrent access synchronizing protocols.

Several integrated database languages have been proposed in recent literature. These include TAXIS [MBW80], Galileo [ACO85] and Amber [Car84]. We will describe some of these briefly in Chapter 3.

2.3.4. Application-specific Database Operations

The semantics of the design database are partly defined by a set of complex, application-specific, user-defined database operations. These operations may be used for sophisticated data manipulation, for computing derived (property) values, and/or for checking complex semantic integrity and consistency constraints. By providing these operations at the database schema level, both programmed access (through design tools) and interactive access are simplified as the operations can be specified and implemented once, and invoked through a simple command or procedure (or function) call. These operations can be specified as mappings between specified value domains [SMF86, Zil84]. They can be implemented in a variety of ways [DKL85]:

(1) Relation tables may explicitly store the graph of the function (set of pairs of corresponding argument (input) and result (output) values) defined by the operation; the operation simply "looks up" the appropriate tuple or tuples in the relation table to compute the desired result. In this case, some form of stored data compression may be desirable to minimize the storage requirements.

(2) A user-defined procedure (or function) written in the high-level, procedural database language may be executed with appropriate arguments to compute the desired result.

(3) A high-level query/data manipulation language statement (or group of statements) may define the result in terms of basic data model operations or other, previously defined, operations.

An example of an application-specific, user-defined database operation is given in Chapter 4.

2.3.5. Database Integrity Constraints

Integrity constraints define valid database states. These constraints can be classified into *inherent constraints* (structural integrity) and *explicit constraints* (semantic integrity). Inherent constraints are a part of the data model's definition. For example, the one-to-many relationships between owner and member records in a CODASYL/DBTG set (network data model) and the uniqueness of tuples within a relation (relational data model) are inherent constraints, and therefore, cannot be violated. For example, the semantic data model of Section 2.3.1 provides referential integrity (all entity-valued references are valid) as an inherent constraint. Explicit constraints are independent of the data model, and are defined outside the framework of the data model's definition. For example, key attributes for a relation scheme, and functional, multivalued etc. dependencies are explicit constraints that can be specified in the relational data model. Integrity constraints can also be classified as *static constraints* (which hold for every database state) or *dynamic constraints* (which specify constraints on update operations). Explicit constraints can be specified as formulas of a first-order logic involving entity variables, constants, comparison and Boolean operators and function symbols. In the relational data model, for example, these can be defined by relational calculus expressions. Other more complex constraints require the full power of partial recursive functions for their specification. These can be implemented by constraint checking procedures (or functions) written in a high-level, procedural language. The execution of these procedures can be

invoked explicitly by the users or "triggered" by an *integrity management subsystem* whenever specified "events" (the database is updated, a new entity or relationship instance is created, or an operation is invoked etc.) occur. The issues involved in designing an integrity management subsystem are complex and beyond the scope of the present work. Hence, in the database language IDL described in Chapters 4 and 5, there is at present no syntax to support the specification and use of integrity checking and maintenance triggers.

Explicit constraints in design databases may be grouped into *value domain, key, class membership, composition, interface, consistency,* and *concurrent access constraints*.

- *Value Domain Constraints* arise from the semantics of the design application. In a VLSI circuit design system, for example, these may include: (1) "geometric design rules" which specify constraints on the cell layout geometry in terms of clearances, sizes, orientations etc. (2) "fan-in" and "fan-out" constraints which specify how many inputs and outputs can be connected to a gate's terminals. There may also be derived value constraints which require that some attribute values are derived from other attribute values.
- *Key Constraints* specify that certain attributes of a class form a key for the entities in the class; that is, the values associated with such key attributes are unique within the class and can be used in entity retrieval operations.
- *Class Membership Constraints* capture the generalization/specialization abstraction in terms of specified interrelationships between entity classes; for example, the *Is-a (subclass of)* relation between classes that defines a generalization/specialization hierarchy of entity classes.

- *Composition and Interface Constraints* arise whenever two or more design entities are "composed" or "interfaced" to form more complex entities. For example, in a VLSI circuit design system, a composition constraint may be that two boxes on the same layer of a layout cell do not overlap. Similarly, an interface constraint may be that ports of some of the components constitute the interface ports of the layout cell being formed.
- *Consistency Constraints* specify mutual consistency requirements among different representations of the same design entity. For example, it is necessary to ensure that the physical layout, the transistor network, the logic circuit and the functional block schematic representations of a given VLSI chip are all mutually consistent and updates to any one of them are properly propagated to the others. For example, Katz and Wiess [KaW83] define that "a VLSI circuit design is consistent if conformance, composition and equivalency constraints are met; that is, (1) a design entity's implementation satisfies its "interface constraints" *(conformance);* (2) the composition of component entities is "well-formed" *(composition)*; and (3) entities specified as equivalent across representations are shown to be equivalent *(equivalency)*."
- *Concurrent Access Constraints* ensure that concurrent accesses to the database by several designers do not result in inconsistencies or conflicting updates. These are usually specified as concurrent access protocols implemented by a *concurrent access control subsystem.*

2.3.6. Concurrent Access Control and Transaction Management

Concurrent access constraints are required to prevent "conflicts" among concurrent user "transactions" trying to update the database simultaneously. A transaction is defined as a sequence of primitive database maintenance operations (INSERT, DELETE and MODIFY) that preserves database integrity and consistency, and whose execution is *atomic* (all operations are executed or none is executed) and *durable* (resilient to hardware or software failures). A *schedule* of concurrent transactions is an ordering on their component primitive database maintenance operations; a *serial schedule* orders the transactions to execute sequentially. The classical concurrent access synchronization problem is to generate a *serializable schedule* of user transactions; i.e. a schedule which produces the same effect on the database as some serial schedule. The well known techniques of *2-phase locking* and *time stamping* provide a solution to the concurrent access synchronization problem. However, these are inappropriate for a CAD database for two reasons:

(1) locking entire entity composition hierarchies may result in large parts of the database being locked; and,

(2) "design transactions" are interactive and, typically, last over extended time periods.

Both these characteristics will preclude any reasonable degree of concurrency if implemented by conventional locking protocols. One solution is to copy design objects, accessed by a user for write/update, to the user's private "data space" (analogous to "address space"), modify the object and finally to copy it back into the database [LoP83]. This leads to the concept of *C-transactions* (for Conversational transactions) defined by "check-out" and "check-in" operations to be implemented by a transaction management subsystem of the design DBMS. Also, the hierarchical structure of design objects necessitates access protocols based on different types of locks on objects under concurrent

access. One suggestion involves two types of locks: an "intentions lock" to indicate that a component object is to be modified and an "exclusive lock" to indicate that no other transaction may access the object till its modification by the locking transaction is complete [Kat82]. Another proposal [Kla85] involves organizing transactions in a two-level hierarchy. A *group transaction* copies desired objects from a "public database" into a "group database." A number of *user transactions* which are created as subtransactions of the group transaction then copy objects from the group database into several private user databases. Updates to database objects are modeled by "version graphs" which show how given versions are derived from other versions. Consistency is implemented by means of different lock types: R (Read only), RD (Read & Derive), DS (Read & Derive shared), DX (Read & Derive exclusive) and X (Read, Derive & Modify exclusive). A "lock compatibility matrix" defines mutual compatibility between these lock types. Transactions request locks on objects before checking them out of the database and release them after checking them back in. While concurrency among group transactions may be regulated by a 2-phase locking protocol (ensuring strict consistency), user transactions may be regulated by a non-2-phase locking protocol (permitting increased parallelism at the cost of some inconsistency). A more detailed study of these issues is beyond the scope of this thesis.

2.4. Summary

In this chapter we have focused on the special requirements of design databases as contrasted with those of conventional databases. We then describe the limitations of traditional database systems in addressing these requirements. Next, we specify the elements of a CAD database system architecture. These elements include: (1) a conceptual-level semantic data model (which captures the concepts of design objects and their interrelationships); (2) a mathematical and logical data model (which provides a formal

specification and implementation layer for the semantic data model); (3) an integrated database language (which eliminates the "impedance" mismatch between the data model and the application program and simplifies specification of complex database operations and semantic integrity and consistency constraints; and (4) a concurrent access control and transaction management subsystem (brief skecth only given herein).

Chapter 3

A SURVEY OF CURRENT RESEARCH IN CAD DATABASES

3.1. Introduction

As described in the Section 2.2.1, the classical data models are inadequate to capture the complexity of structures, relationships and behaviour found in a design database. Several extensions to the classical data models as well as a variety of *semantic data models* have been proposed to conceptualize these requirements. These include the *Binary Data Model* [Abr74], the *Semantic Data Model* (SDM) [HaM81], the *Entity-Relationship Data Model* [Che76], the *Semantic Hierarchy Model* [SmS77], *RM/T* [Cod79], the *TAXIS* data model [MBW80], and the functional data models ([BuN84] and *DAPLEX* [Shi81]). A semantic data model provides primitives to model entities with complex structures, and "typed relationships" among these entities. Sowa [Sow81] lists the requirements to be supported by a conceptual schema for "knowledge-based systems" to include aggregation hierarchies, functional dependencies, abstract domains, procedural attachments and inference mechanisms. McLeod and Smith [McS81] view a data model as defining (1) a "data space" consisting of atomic elements and relationships among them; (2) "type definitional constraints"; (3) data manipulation operations; and (4) a predicate language that enables data selection and retrieval operations.

3.2. Augmenting Classical Data Models

The approach of augmenting classical relational or entity-relationship data models involves adding to the basic data models additional concepts, abstraction mechanisms and operators especially relevant to the application domain. GEM [Zan83] is a data model that extends the relational data model with the concepts of entities (with surrogates), set-valued attributes, generalization and aggregation abstractions, null values and

an enhanced QUEL-based query language. A GEM *entity* is defined in terms of an entity name, a sequence of attribute values and a key attribute set. An attribute value may be atomic (integer, real, etc.), a set of values, a reference to another entity, or a tagged value with the tag indicating one of a list of specializing value classes. Entities can be accessed through their key attribute values, or through their surrogates which are system-generated unique identifiers. The QUEL-based query language provides the notions of "functional composition" and (explicit and implicit) "entity joins" which enable complex retrieval operations to be specified. A 3-valued logic with TRUE, FALSE and NULL truth values is defined to handle undefined attribute values.

Batory and Kim [BaK85] define data modeling concepts of particular relevance to VLSI design applications. In their conceptual model, a design object is characterized by an interface description *(type)* and one or more implementation descriptions *(versions)*. Primitive objects have only interfaces but no implementations. A *molecular object* is an aggregation of a set of simpler component objects. A version of a molecular object is defined by specifying a set of component object types and interconnections between them, and a set of mappings between the interfaces of the molecular object and its component types. An *instance* of an object type or an object version is a copy which inherits attributes from the object type or the object version. This eliminates redundancy in stored data by abstracting common information among a number of instances into the object type or version attributes. Object types, versions and instances can also have "external features" associated with them. A *parameterized version* is a molecular object with component object types (interfaces), rather than versions (implementations), defined. Thus, these objects can be thought of as consisting of "sockets" (defined by object types); the molecular object can be implemented by "plugging" in any appropriate component object versions. The model also provides for structuring object types, versions and instances in

terms of entities and relationships of the Entity-Relationship model (at the conceptual level) and mapping such structures into relation schemes (at the internal level). A set of basic operations on types, versions and instances is also defined.

Hardwick [Har84] proposes an enhanced relational model based on *homogeneous* and *heterogeneous* relations which correspond, respectively, to aggregation and generalization abstractions. An enhanced algebra provides data model operations of *AND* and *OR integrations* (relation definition), *projection, enclosure* (abstracting a relation to form an attribute), *disclosure* (inverting an enclosure to form a relation), *differentiation* (searching for values belonging to an attribute in a hierarchy) etc. It also provides a simple method of constructing user-defined functions and applying "join" and "selection" operations to such functions.

3.3. Semantic Data Models

In this section we describe some of the many semantic data models that have been proposed in the literature. Brodie [Bro81] describes a database system called *ACM/PCM (Active Component Modeling/Passive Component Modeling)* which is based on an extended semantic data model. This data model provides "hierarchies of multiply typed data abstractions", a "data type algebra" and "procedure control abstractions." An *object scheme* defines structural aspects of the database including objects and relationships while an *action scheme* defines the behavioral aspects in terms of operations applicable to these objects. Semantic integrity is ensured by specifying all data manipulation through operations defined in the action scheme. This database system has been used in the design of Criminal Court and Real Estate information systems.

SDM (Semantic Data Model) [HaM81] conceptualizes a database as a collection of *entities* (representing corresponding entities in the real world); these entities are grouped

into *entity classes* which can be related to each other by means of *interclass connections*; entities and entity classes have *attributes* whose values are drawn from specified "value domains". An entity class can be a *base class* (defined independently of all other classes) or a *nonbase class* (defined in terms of one or more other classes). The interclass connections include the *subclass* connection and the *grouping* connection. A set of rules define attribute and attribute value inheritance across these interclass connections.

Buneman and Nikhil [BuN84] describe a conceptual data model based on the mathematical notion of functions. The *functional data model* defines a database as a collection of data types and functions over these types. For example, the relation scheme:

EMPLOYEE (NAME:string, SS#:integer, FULLTIME:boolean)

is regarded as a set of functions,

NAME: EMPLOYEE → string
SS#: EMPLOYEE → integer
FULLTIME: EMPLOYEE → boolean

A set of functional operators (extension, restriction, construction and composition) enables powerful, applicative expressions of high-level operations on the database. For example, the expression,

! EMPLOYEE . * NAME

defines a composition (.) of a function that generates a list of EMPLOYEE entities (! EMPLOYEE) with a function that returns their names (* NAME). They suggest that the combination of a simple data structure (entity set) and associated operations, the notion of function application and composition, and a user-definable type system provide a powerful database processing environment. The functional data model's query language *FQL (Functional Query Language)* is an applicative (or functional) language where the only control structure is the application of functions to arguments (Lisp-like expression evaluation) based on the functional data model. It is claimed that besides providing a

more powerful database language than relational algebra, FQL provides a simple, uniform and consistent way of specifying user-defined types (strong typing and abstract data types), and user-defined "high-level operations" (through functional composition).

3.4. Interfacing Data Models to Programming Languages

Current approaches to the problem of interfacing data models to programming languages draw heavily on the techniques of abstraction (data and procedural) in programming language theory. One approach to embedding database operations in programming language constructs is presented by Mall, Schmid and Reimer [MSR84]. The basic idea is to define relations as "typed and named objects" which can be imported into the scope of a "database processing environment." A *generalized selector* mechanism based on first-order predicate calculus expressions provides for relation element selection operations to be defined. A set of selection, assignment and update operations are also provided. The database operations considered include access and integrity control, failure recovery, data sharing, and concurrency management. Concurrent access control is achieved by visibility control (modularization of scopes and import/export clauses) and access modes (read, write, readwrite) of selected relation variables. Transactions are defined as sequences of operations that preserve database integrity and consequently must be viewed as "atomic actions." A transaction must, therefore, be executed to completion, or in case of failure, aborted and the database restored to its state at the start of transaction. Concurrent execution of transactions operating on overlapping objects can be achieved by a variety of locking protocols applied at different locking granularities determined by selection predicates. These concepts have been implemented in Pascal/R [ScM80] and Modula/R [Rei84].

Modula/R extends the Modula-2 language with database constructs derived from the relational data model. The data model objects are *relations* and *relation elements*

(individual tuples or subsets of a relation). In addition to the basic operations of relational definition, assignment and update, the data model operations include a *generalized selector* mechanism which is a first-order predicate calculus expression for specifying relation element selection. Concurrent access to database objects is controlled by employing the Modula features for specifying variable *scopes* (visibility) and *access modes* (Read, Write and ReadWrite). The *transaction* defines the unit of integrity, and concurrent transaction access is handled by locking database objects as determined by selection predicates. Thus, transactions accessing the same relation but non-overlapping selected elements can be executed in parallel.

Smith, Fox and Landers [SFL83] describe the design of ADAPLEX, a system for developing database application programs. It extends the programming language Ada to include database constructs derived from a database sublanguage based on the functional data model DAPLEX [Shi81]. The data model objects consist of *entities* and *entity sets*. The abstraction mechanisms include entity *subtype* and entity *supertype* which model, respectively, specialization and generalization abstractions for entity types. The constraints on the data model objects are specified as *consistency rules* defining *overlap* and *uniqueness* constraints for entity sets. The basic database operation is the *transaction* which encapsulates all database access operations and is defined to be the unit of concurrency, consistency and failure recovery. The specifications are preprocessed to generate a runtime database module and a pure Ada program (with database function calls in it) which can then be compiled and executed.

3.5. Integrated Database Languages

GLIDE2 (Graphical Language for Interactive DEsign) is an integrated programming language and database system designed to support interactive design applications [EaT79]. It was designed to efficiently support interactive operations on dynamically structured data. A GLIDE database is structured in terms of "frames" (an extensible context or workspace), "objects" (constants, types, variables, procedures and modules), "structures" (array, record, set, file and vector) and user-defined abstract data types. Its features include management of large amounts of data (10^7 to 10^{12} bytes), a dynamically definable database schema, geometric modeling primitives, fast access to data on secondary storage, complex integrity constraint management, multi-user concurrent access, and backup and recovery facilities.

TAXIS [MBW80] is a language for designing and implementing "interactive information systems." It integrates concepts from Artificial Intelligence, Databases and Programming Language areas. Its design objectives were to provide a conceptual-level, semantic data model, complex transaction definition and exception handling facilities, and a compilable specification language. A TAXIS database is structured in terms of *objects* and *properties*. An object may be a *token* (atomic value), a *class* (set of tokens) or a *metaclass* (set of classes). Properties are triples: *<subject, attribute, p-value>* where *subject* and *p-value* are objects and *attribute* is a name. Token-valued subject and p-values define *factual properties* (structure of the token specified by the subject) and class-valued subject and p-values define *definitional properties* (structure of "instances" of the subject class). Classes can also be specified as instances of a "variable-class" metaclass (the set of tokens associated with these classes is variable), a "finitely-defined-class" metaclass, a "formatted-class" metaclass etc. TAXIS classes and metaclasses are organized into an *inheritance hierarchy* defined by the binary *is-a* relationship. If A is-a B,

then every instance of A is also an instance of B and A inherits all the properties of B; A can have additional properties or redefine some of B's properties. Relationship *IS-A* is defined as the reflexive and transitive closure of the is-a relationship. The definitions in a TAXIS schema have to satisfy the "IS-A relationship postulates." The set of data model or primitive operations include insert object into a class, delete object from a class, iterate over the instances of a class, and update a property value of an object. Transactions (groups of primitive operations) are also modeled as classes whose instances are the values returned by an execution of the transaction. Transaction classes are defined in terms of prerequisite, action, result and return value properties. The corresponding p-value classes are "expressions" which define computations on objects in terms of arithmetic, assignment and data model operations, and conditional, block and looping control flow constructs. The transactions classes and expressions are also conceptually organized into the same *is-a* hierarchy as other classes, thus providing a single unifying framework for structuring both data and computations.

Galileo [ACO85] is a high-level, interactive, strongly-typed, expression-oriented conceptual language supporting semantic data model features. The data model provides a type system consisting of *concrete* (primitive) and *abstract* (concrete types augmented with constraints and operators) *data types, subtypes, entity class* and *subclasses,* and *tuple, sequence* and *set* aggregation abstractions. The language also provides the concept of an *environment* which is simply a group of expressions which define a set of data structures and operations by means of an "environment expression". This is a modularization feature enabling incremental development of a database system. The system also provides facilities to specify groups of expressions as *transactions* which are units of semantic integrity and failure recovery.

Amber [BuA86] provides a set of parametrizable data types, supports inheritance on

types and a very general form of persistence. It also supports a form of polymorphism in which values of a most general type - *dynamic* - can be "coerced" to values of more specific types. However, this approach is likely to be very inefficient because there is no clustering of values by type and it requires elaborate functions and control mechanisms for creating, inserting and testing values in the database.

3.6. Object-oriented Database Management Systems

Several database system implementations based on object-oriented concepts have been proposed. Gibbs [Gib84] presents an object-oriented data model for office information systems. The *object-oriented office data model* is based on the concept of modeling the structures and semantics of common office objects like stations, desks, files, cabinets, clocks and calendars. The model defines the structures (*object, data type* and *template type*), operations and constraints (*domains* and *triggers*) applicable to these objects.

Maier and Copeland [CoM84] describe a database system based on the concept of integrating a "set-theoretic" data model with the object-oriented programming environment of Smalltalk. The conceptual facilities of importance to database systems are defined to include a flexible and powerful type system (ability to define new types, add operations to existing types and separate data declaration from type definition), modeling entity identity, and data abstraction mechanisms. In this context, useful Smalltalk features include the notion of *objects* (which provide entity identity), the *class mechanism* (which provides the data abstraction and type definition facility) and *messages* (which provide operations on objects). In the set-theoretic data model, objects are viewed as *labeled sets*. Each set element has a *name* (label) and a *value* (which can be another labeled set). A "path-syntax" for accessing component parts of a set and a set-calculus query language are also defined. A combination of Smalltalk and the set-theoretic data model augmented with authorization, concurrency control and recovery

subsystems provides a design for a complete database system.

IBM is developing a database system for VLSI design [DiL85] which involves building an object-oriented system on top of an underlying relational system. The DBMS provides the notion of *complex objects* (which model design objects) which are system-implemented groups of related tuples. Tuples (from different relations) forming a complex object are related by using system-generated *tuple IDs* as values for attributes defined over abstract domains COMPONENT_OF and REFERENCE which implement the corresponding relationship abstractions. This enables users to specify database operations like fetch, delete, copy, lock etc. on objects and the system automatically translates them into appropriate operation sequences on the underlying relations. The complex objects and abstract domains also enable implementation of referential integrity, multiple versions and design transactions. A VLSI layout editor based on this database system has been implemented [HaL84].

3.7. Comparative Analysis and Summary

In this chapter, we have presented an overview of current research on issues of relevance to the design of D-DBMSs. These research efforts have involved augmenting classical data models with additional abstraction mechanisms, designing semantic data models tailored for specific classes of applications, interfacing data models to programming languages, designing integrated database languages, and using object-oriented models and techniques in the design and implementation of DBMSs.

Table (3.1) tabulates the design database system features required, and shows how existing and proposed systems evaluate in the context of these requirements. We compare four classes of models/systems and typical representatives from each of these:

Augmented and Extended Relational Models

This class represents relational systems extended with the notion of entity and some of the required abstraction mechanisms. For example, GEM [Zan83] provides *surrogates* which implement entities, and limited forms of Cartesian and Set aggregation and generalization. While it simplifies and enhances a QUEL-like relational calculus based query language, it does not provide the features of classification, structural integrity constraints, entity interrelationships and complex operations or semantic integrity and consistency constraints.

General-purpose Semantic Data Models

This class consists of general-purpose semantic data models based on the notions of entities and entity classes. For example, SDM [HaM81] provides the basic structural abstractions of classification, generalization/specialization etc. but not the entity interrelationships, complex operations or semantic integrity constraints required in a design context.

Functional Data Models

This class of systems is based on the notion of mathematical functions as the basic data structuring construct. DAPLEX [Shi81] was initially proposed as a high-level, user-friendly interface to an underlying relational, network or hierarchical database. ADAPLEX [SFL83] which extends ADA for database applications by embedding the DAPLEX sublanguage comes closest to satisfying most of the essential features in a design database, but lacks sequence aggregation, entity interrelationships and provides only limited generalization abstraction and integrity constraints.

NF^2 Relational Data Model

A number of systems based on non-first normal form of the relational data model have been proposed in recent literature [AMM82, Dad86, JaS82, PiA86]. These seek to

remove the restrictions imposed by normalized relation schemes when the data being modeled does not inherently have a tabular structure. However, the NF^2 relational model still lacks the basic notion of entities and entity classes. So, although it provides Cartesian, Set and Sequence aggregation through values that can be tuples, sets or sequences, the ease and flexibility with which such values can be manipulated is severly limited. For example, lacking the notion of entity identities these complex structures (values) cannot be shared, passed as parameters or otherwise processed as entities with their own identities. The NF^2 model also does not provide structural or semantic integrity constraints, nor entity interrelationships and complex operations.

In capturing these requirements, the notion of system-defined as well as user-defined ADTs is very useful. For example the notions of entities and their interrelationships, the aggregation abstractions etc. can be modeled by appropriate ADTs. However, none of above classes of database models/systems provides a useful set of type constructors and general Turing procedures which can be used to define ADTs. Many recent database systems and languages seek to remedy this defect as discussed in detail in Chapter 5.

CAD Database System Features: Comparative Analysis Table					
No.	Feature	Augmented Relational Data Model	General-purpose Semantic Data Model	Functional Data Model	*NF*² Relational Data Model
1.	Entity	Y	Y	Y	N
2.	Classification	N	Y	?	N
3.	Aggregation				
3.1.	Set Aggregation	Y	N	Y	Y
3.2.	Sequence Aggregation	N	N	N	Y
3.3.	Cartesian Aggregation	Y	Y	Y	Y
4.	Generalization-Specialization	restricted	Y	Y	N
5.	Constraints				
5.1.	Domain Constraint	N	N	Y	N
5.2.	Key Constraint	Y	N	Uniqueness Constraint	?
5.3.	Class Membership Constraint	N	Y	Overlap Constraint	
6.	Entity Interrelationships				
6.1.	Representation Relationship	N	N	N	N
6.2.	Version Relationship	N	N	N	N
6.3.	Instance Relationship	N	N	N	N
7.	Data Model Operations	Y	Y	Y	Y
8.	Structural Integrity				
8.1.	Entity Uniqueness	?	Y	Y	N
8.2.	Referential Integrity	?	Y	N	N
8.3.	Entity Typing	N	Y	Y	N
9.	Database Operations				
9.1.	Query Language	Y	N	Y	Y
9.2.	Tuning Procedure	N	N	?	N
9.3.	ADT operations	N	N	?	restricted
9.4.	Triggers	N	?	N	N
10.	Derived Values	N	Y	Y	N
11.	Complex Semantic Integrity Constraints	N	N	Y	N

Table (3.1) CAD Database System Features: Comparative Analysis

Chapter 4

A SEMANTIC DATA MODEL FOR DESIGN DATABASES

4.1. Introduction

As we have seen in Section 2.2, a range of structural and behavioural abstraction mechanisms, including referential integrity, structural aggregation and procedural abstractions, are required in modeling a design database. A semantic data model seeks to capture directly (in terms of the data model constructs) the application domain concepts. In a CAD application, the semantic data model has to provide constructs to correspond to concepts like entity, entity class, entity interrelationship, composition hierarchy etc. These can be implemented directly, or indirectly through another level that provides its own set of (mathematical/logical) constructs. In this chapter we provide a formal specification of our semantic data model which includes the notions of *entities, entity classes, entity interrelationships, integrity constraints,* and *operations*. Illustrative examples from the domain of VLSI design are also given to show how the design database requirements are captured by the model.

4.2. Entities

Entities are data which model real world objects in an application. A typical design process will have pre-existing entities and will create more entities. Their specification occurs during a design process. This notion of entity corresponds to the notion of entity as defined by the Entity-Relationship model [Che76]. In a VLSI design system, the data describing *gates, chips, functional blocks, transistors* etc. form the design entities. Each entity has a unique *identifier* (*surrogate*) which is a system-defined value drawn from a suitable value domain, and a set of associated properties defined in terms of *attribute values, constraints* and *relationship instances*. Entities can be implemented as instances

(values) of an abstract data type *Entity*. User access to the entities is provided by user-specified *variables* (which are bound to specified entities by the system), and by selection operations (of the data model) which specify key attribute values of entities (e.g. its user-defined name) or its relationship to other entities (e.g. "the amplifier in C-145"). A design system must have such entity identities since an object may not have enough property values to uniquely identify it.

4.3. Entity Classes

Entity classes provide the classification abstraction.

Definition 4.1: An *entity class intension* is the collection of properties (*attributes* and *constraints*) shared by all the entities belonging to the class.

Definition 4.2: If E is an entity class, we define the *domain* of class E (denoted dom(E)) as the set of all entities which share the properties defined by the intension of class E:

$$\mathrm{dom}(E) = \{\ e \mid \text{entity } e \text{ has at least properties defined by the intension of class } E\ \}$$

Definition 4.3: The *entity class extension* (denoted *Ext(E)*) is the set of all entities, stored in the database at a given time instant, that posses the properties defined by the corresponding entity class intension:

$$\mathrm{Ext}(E) = \{\ e \mid e \in \mathrm{dom}(E) \text{ and } e \text{ is stored in the database}\ \} \subset \mathrm{dom}(E)$$

Entity class names are unique and distinct class names denote distinct classes. Examples of entity classes in a VLSI design database are data representations named *Cell, Box, Gate, Chip* etc. All entities in the database have the system-defined abstract type *Entity*, and collectively , can be viewed as constituting the extension of a most general *Entity* (or *Object*) class.

4.4. Attributes

Attributes specify properties of entities belonging to an entity class. Each attribute has an associated *value type*. The value type can be an *atomic type* (entity class or primitive data type {Integer, Real, Boolean, String}) or a *composite type* (tuple, set or sequence type whose values are tuples, finite sets or sequences of values of atomic or composite types). Thus, an attribute can be visualized as a mapping from the entity class to the domain of its associated value type.

Definition 4.4: If entity class E defines an attribute A with value type T, then we have the time-varying mapping:

$$M[A]: Ext(E) \rightarrow dom(T)$$

where *dom(T)* denotes the domain of values of type T. A special value NULL $\in$ *dom(T)* may be used to indicate that an entity in class E has no value assigned for attribute A at a particular time instant. Thus, if attribute A of entity e has a value v (not equal to NULL), then:

$$M[A](e) = v$$

If the value of attribute A of entity e is undefined, then:

$$M[A](e) = NULL$$

For basic data types T, dom(T) is defined in the usual set-theoretic sense; for composite types, set type $S = \{T\}$ and sequence type $Q = \langle T \rangle$, we define dom(S) as the set of all finite sets of values of type T, and dom(Q) as the set of all finite sequences of values of type T respectively. Thus, an entity attribute has a value which may be: (1) a basic data type value (Integer, Real, Boolean and String value, which may viewed as a primitive entity); (2) another entity in another (not necessarily distinct) entity class; or, (3) a structure (tuple, set or sequence of entities or basic data type values).

Thus, the collection of attributes in an entity class intension implements the Cartesian

aggregation abstraction by mapping each entity in the entity class extension into its "component" entities. Attribute values which are structures (tuples, sets and sequences) provide the *tuple, set* and *sequence aggregation.*

Definition 4.5: The function *Attrib* returns the set of attributes defined for (i.e. applicable to members of) class E:

$$\text{Attrib}(E) = \{ A \mid \text{attribute } A \text{ with value type } T \text{ is defined for class } E \}$$

Definition 4.6: The attributes of a collection of entity classes also define the *HasComp* relationship (aggregation abstraction) specified as:

$$HasComp = \{ (E_1, E_2) \mid \exists A \in \text{Attrib}(E_1) \text{ such that } A \text{ has value type } E_2, \{E_2\} \text{ or } \langle E_2 \rangle \}$$

The transitive closure of *HasComp* is denoted by *HasComp**.

Example 4.1: If entity class *Cell* defines attributes *cell_id, cell_type, ll_corner, ur_corner* and *struct* with, respectively, types *Integer, String, Point, Point* and {*Element*} then we have:

Attribute functions:
M[*cell_id*]: dom(*Cell*) → dom(*Integer*)
M[*cell_type*]: dom(*Cell*) → dom(*String*)
M[*ll_corner*]: dom(*Cell*) → dom(*Point*)
M[*ur_corner*]: dom(*Cell*) → dom(*Point*)
M[*struct*]: dom(*Cell*) → dom({*Element*})

Attribute set:
Attrib(*Cell*) = { *cell_id, cell_type, ll_corner, ur_corner, struct* }

Entity composition hierarchy:
(*Cell*, *Point*) ∈ *HasComp*
(*Cell*, *Element*) ∈ *HasComp*

For notational simplicity, we will henceforth write *A(e)* for *M[A](e)*.

4.5. Constraints

Our data model provides for three types of data model integrity constraints.

4.5.1. Domain Constraint

A domain constraint defines the set of valid values for specified attributes or groups of attributes.

Definition 4.7: A *domain constraint* is specified as a first-order predicate calculus expression composed of attributes, value constants, comparison and Boolean operators. The domain constraint constrains attribute values of entities in the class such that a substitution of an entity's attribute values for the corresponding attributes in the expression evaluates to the Boolean constant TRUE or the value NULL. An arithmetic or comparison operation involving NULL values evaluates to a NULL; Boolean operations involving NULL values evaluate according to a 3-valued logic defined by the truth table given below:

3-valued Logic				
v1	v2	v1 AND v2	v1 OR v2	NOT v2
T	T	T	T	F
T	F	F	T	T
F	F	F	F	T
T	NULL	NULL	T	NULL
F	NULL	F	NULL	NULL

Table 4.1 3-valued Logic

We define $(Pred)[A(e)/A]$ to denote the expression obtained by substituting $A(e)$ for all A in the expression $Pred$. Thus, if D is a domain constraint with associated constraint expression $Pred$ defined by class E, then for $e \in \text{Ext}(E)$, if $(Pred)[A(e)/A] = \text{TRUE}$, then the constraint D is satisfied for entity e; else if $(Pred)[A(e)/A] = \text{NULL}$, then the constraint D cannot be evaluated for entity e. If the domain constraint expression evalu-

ates to FALSE, the domain constraint is not satisfied, and the system will disallow specification of an attribute value that causes such a violation of the constraint.

Example 4.2: The entity class *Cell* may define a domain constraint D to be ($1 \leq$ *cell_id* & *cell_id* ≤ 4000), which constraints *cell_id* attribute values to lie in the specified range.

4.5.2. Key Constraint

Definition 4.8: (Key constraint) A given subset of attributes defined by an entity class forms a key for entities in the class.

The attribute values associated with the key attributes of each entity in the class are unique and can be used to select a unique entity from an entity class in a selection operation. Since entities are internally managed through system-defined identifiers, a user-defined key attribute may be undefined (i.e. NULL valued). The key constraint only ensures uniqueness when all key attributes are defined. Mandatory non-NULL keys would require the key values to be specified at entity creation time and no update of the key values would be permitted. Thus, if A_i (i = 1, .., k) are the key attributes defined by class E, then ($\forall e \in \text{Ext}(E)$) the tuple ($A_1(e)$, .., $A_k(e)$) is unique within the class E provided all the values $A_i(e)$ are defined (i.e. not NULL).

Example 4.3: The *Cell* entity class may define a key constraint K to be K = { *cell_id*, *cell_type* }, which defines *cell_id* and *cell_type* to be key attributes.

4.5.3. Class Membership Constraint

A class can be specified to have a set of *subclasses* or be formed by a union of other classes.

Definition 4.9: The subclass/superclass relationship between classes is captured in terms of the *Is-a (subclass of)* relation defined as:

$Is\text{-}a$ = { (E_1, E_2) | E_1 is a subclass of E_2 or E_2 is formed by a union of other classes including E_1 }

The transitive closure of the $Is\text{-}a$ relation is denoted by $Is\text{-}a^*$. A subclass inherits properties in its superclasses and defines additional properties. This can be modeled in terms of an injection (I_1) from a subclass (E_1) to a superclass (E_2):

I_1: dom(E_1) → dom(E_2)

which maps an entity in the subclass to its corresponding "image" in the superclass. This injection can be implemented by inserting every newly created entity belonging to a class E into the extension of E and, recursively, into the extensions of all its superclasses in the *Is-a* hierarchy. This induces the following *class membership constraint.*

Definition 4.10: For all classes E_1, E_2 such that $(E_1, E_2) \in Is\text{-}a$ (Ext(E_1) ⊆ Ext (E_2))

This allows us to view entities at different levels of abstraction corresponding to the different classes they are found in. We also define a "most general" class *Entity* whose extension contains all entities as members. This is modeled by an injection (I_2) from every class (E) to the class *Entity*:

I_2: dom(E) → dom(*Entity*)

which maps every entity in class E to its corresponding "image" in the class *Entity*. This can be implemented by means of an abstract data type *Entity* associated with every entity (value).

Example 4.4: Consider the specification of *Cell* and *CellInstance* classes given in Example 4.9. The *CellInstance* entities represent particular instantiations of *Cell* entities, and hence can themselves be viewed at another level of abstraction, as cells. Thus:

(CellInstance, Cell) ∈ Is-a
Ext(CellInstance) ⊆ Ext(Cell)

When the system creates a *CellInstance* entity by inserting a newly created entity into the extension of the *CellInstance* class, it uses the *Is-a* hierarchy to also insert the entity into

the extension of the *Cell* class. Now, the entity has the attributes "instance#" (instance-id) and "transform" (instance transformation matrix) by virtue of being in the *CellInstance* class and the attributes "cell-id", "cell-type", "ll-corner", "ur-corner" and "component" by virtue of being in the *Cell* class. This allows us to view an entity at different abstraction levels; in this case both as cell instance and as a cell. The many-to-1 mapping from cell instances to cells can be captured by making a set of entities with different "instance#" and "transform" values (in the *CellInstance* class) have identical "cell-id", "cell-type", "ll-corner", "ur-corner" and "component" values (in the *Cell* class). An alternative formulation that uses the *Instance-of* entity interrelationship is given later in Example 4.8.

4.6. Entity Interrelationships

A set of "typed" relationship relations implement entity interrelationships. *Representation* relations define the relationships between entities which are different representations (views) of the same abstract design object (multiple representation abstraction). *Version* relations define relationships between alternative implementations (*implementation versions*) or updates (*update versions*) of a generically defined entity (multiple version abstraction). *Instance* relations define the instance/master relationships between entities (multiple instance abstraction). Associated with each relationship is a set of *operations*, a *cardinality constraint* and *protocols* (functions/procedures that implement the associated semantic constraints) which together define the semantics of the relationship.

4.6.1. Representation Relationship

A design entity may have several different representations at different levels of abstraction together with semantic constraints to ensure mutual consistency among them. In our model, the multiple representation abstraction is captured in terms of a set of "named" (or indexed) representation relations. The name of the relation determines the meaning (to the designer) of the representation.

Definition 4.11: A representation relation R_i defined as:

$$R_i = \{ (i, e_1, e_2) \mid e_1 \in \text{Ext}(E_1), e_2 \in \text{Ext}(E_2), \text{and entity } e_1 \text{ is a "}i\text{ representation" of entity } e_2 \} \subseteq \{i\} \times \text{Ext}(E_1) \times \text{Ext}(E_2)$$

models the representation relationship between entity classes E_1 and E_2. Here i is the name of the representation relationship that captures the user-defined application-specific meaning for the entity interrelationship. The relation *Repr-of* specified in Definition 2.6 is now defined by the many-to-1 mapping from $\text{Ext}(E_1)$ to $\text{Ext}(E_2)$ defined by R_i. The associated cardinality constraint given by $c_i = \langle n, 1 \rangle$ specifies the cardinality of this mapping. Note that this is a time varying relation since relationship instances and entities may be created and/or deleted at specified times.

Example 4.5: If we define a *Layout* representation relationship between *Cell* and *Gate* entities, then:

$$R_{Layout} = \{ (Layout, c, g) \mid c \in \text{Ext}(Cell), g \in \text{Ext}(Gate), \text{and entity } c \text{ is a "Layout" representation of entity } g \} \subseteq \{Layout\} \times \text{Ext}(Cell) \times \text{Ext}(Gate)$$

$$c_{Layout} = \langle 3, 1 \rangle$$

4.6.2. Version Relationship

The multiple version abstraction can be defined in several different ways. *Implementation versions* are alternative implementations of a functionally specified entity. *Update versions* are "refinements" obtained by successive updates to a given entity. Again, there are associated semantic constraints to ensure mutual consistency among these versions. In our model, the version relationship instance between entities can be created (or deleted) explicitly by the designer, or by means of operations executed by the system.

Definition 4.12: A version relation V_i defined as:

$$V_i = \{ (i, e_1, e_2) \mid e_1 \in \text{Ext}(E_1), e_2 \in \text{Ext}(E_2), \text{ and entity } e_1 \text{ is a "}i\text{ version" of entity } e_2 \} \subseteq \{i\} \times \text{Ext}(E_1) \times \text{Ext}(E_2)$$

models the version relationship between entity classes E_1 and E_2. Here i is the name of the version relationship that captures the user-defined application-specific meaning for the entity interrelationship. The mapping *Ver-of* specified in Definition 2.7 is now defined by the many-to-1 mapping from $\text{Ext}(E_1)$ to $\text{Ext}(E_2)$ defined by V_i. The associated cardinality constraint given by $c_i = \langle n, 1 \rangle$ specifies the cardinality of this mapping. Note that this is a time varying relation since relationship instances and entities may be created and/or deleted at specified times. A *version history* (or chronology) of update versions can be formed by including a time parameter as in:

$$V_i = \{ (i, e_1, e_2, t) \mid e_1 \in \text{Ext}(E_1), e_2 \in \text{Ext}(E_2), \text{ and entity } e_1 \text{ is a "}i\text{ version" of entity } e_2, \text{ created at time } t \} \subseteq \{i\} \times \text{Ext}(E_1) \times \text{Ext}(E_2) \times \text{dom}(\mathit{Time})$$

Example 4.6: Consider two alternative implementations (classes Add_1 and Add_2, and entities a_1 and a_2) of a full adder (class *FullAdder* and entity f). The entities a_1 and a_2 have different internal structures but the same functionality (interface), presumably captured by the specification of entity d. Entities a_1 and a_2 now represent implementation

versions of entity d. This version relationship can be captured in terms of the version relations V_{Add_1} and V_{Add_2} as follows:

$$a_1 \in \text{Ext}(Add_1),\ a_2 \in \text{Ext}(Add_2),\ \text{d} \in \text{Ext}(FullAdder)$$
$$(Add_1, a_1, d) \in V_{Add_1}$$
$$(Add_2, a_2, d) \in V_{Add_2}$$

Example 4.7: As another example, consider the the relationship between two ALU versions (a_1 and a_2) obtained by updating the ALU (a) at times t_1 and t_2. This can be defined in terms of version relation V_{ALU}:

$$a, a_1, a_2 \in \text{Ext}(ALU)$$
$$(\text{ALU}, a_1, a, t_1), (\text{ALU}, a_2, a, t_2) \in V_{ALU}$$

4.6.3. Instance Relationship

A design may consist of multiple "instances" of a given design entity. Each instance may be just a copy of the original (as when a standard part is used as a component in several different contexts in the design) or a specific instantiation of a master entity with its own instantiation parameters (as when a standard cell is used in different locations with different orientation and scale factors in a chip layout).

Definition 4.13: An instance relation I_i defined as:

$$I_i = \{\ (i, e_1, e_2) \mid e_1 \in \text{Ext}(E_1),\ e_2 \in \text{Ext}(E_2),\ \text{and entity } e_1 \text{ is a "}i\text{ instance" of entity } e_2\ \}$$
$$\subseteq \{i\} \times \text{Ext}(E_1) \times \text{Ext}(E_2)$$

models the instance relationship between classes E_1 and E_2. Here i is the name of the instance relationship that captures the user-defined application-specific meaning for the entity interrelationship. The mapping *Inst-of* specified in Definition 2.8 is now defined by the many-to-1 mapping from $\text{Ext}(E_1)$ to $\text{Ext}(E_2)$ defined by I_i. The associated cardinality constraint given by $c_i = \langle n, 1 \rangle$ specifies the cardinality of this mapping. Note that this is a time varying relation since relationship instances and entities may be created and/or deleted at specified times.

Example 4.8: The instance relationship between a cell c and its instantiations i_1, i_2 can be defined in terms of an instance relation $I_{Geometry}$:

$c \in$ Ext(*Cell*); $i_1, i_2 \in$ Ext(*CellInstance*)
(Geometry, i_1, c), (Geometry, i_2, c) $\in I_{Geometry}$

4.6.4. Other Relationships

A fourth general relationship type (n-ary relation scheme) is also provided. As in the relational model, this has no particular system-defined semantics, constraints or protocols.

Definition 4.14: An n-ary relation G_R is defined as:

G_R = { (v_1, ..., v_n) | values or entities v_1, ...v_n are mutually related by relationship R }

Example 4.10: Consider the entity class, relationship and operations specifications for a VLSI circuit layout database defined in an informal "Schema Definition Language" below.

```
/* entity class specifications */

CLASS Cell = (                  /* layout cell */
 ATTRIBUTE
  cell_id: INTEGER,             /* cell identifier; modelled here as an integer
                                   value, it can also be modelled more generally
                                   as, say, a string value  */
  cell_type: STRING,            /* cell type */
  ll_corner: Point,             /* lower left corner of the cell's bounding box
                                   in some reference coordinate system */
  ur_corner: Point,             /* upper right corner of the cell's bounding box */
  component: {Element};         /* set of component elements of the cell */
 CONSTRAINT
  c_1: RANGE(1 <= cell_id & cell_id <= 4000),        /* domain constraint */
  c_2: RANGE(cell_type IN { "NMOS", "CMOS" });       /* domain constraint */
  c_7: SUBCLASS(CellInstance);                       /* specialization abstraction */
)

CLASS Point = (                 /* layout point */
 ATTRIBUTE
  x: INTEGER,                   /* x coordinate */
```

```
 y: INTEGER;                          /* y coordinate */
 CONSTRAINT
  c_3: RANGE(1 <= x & x <= 1024),/* domain constraint */
  c_4: RANGE(1 <= y & y <= 1024);/* domain constraint */
)

CLASS Element = (                          /* layout element */
 CONSTRAINT
  c_5: UNION (Geometry, CellInstance);   /* specialization abstraction (subclasses) */
)

CLASS Geometry = (                         /* layout geometry */
 CONSTRAINT
  c_6: UNION (Box, Port, Wire);          /* specialization abstraction (subclasses) */
)

CLASS CellInstance = (                     /* a cell (placement) instance */
 ATTRIBUTE
  instance#: INTEGER,                  /* instance id */
  transform: <INTEGER>;                /* placement transformation matrix relative to
                                       the reference coordinate system */
)

CLASS Box = (                              /* layout box */
 ATTRIBUTE
  box_id: INTEGER,                     /* box identifier */
  length: INTEGER,                     /* box length */
  width: INTEGER,                      /* box width */
  layer: STRING,                       /* layer on which the box is placed */
  location: Point;                     /* location of the box in the cell */
 CONSTRAINT
  c_8: RANGE(1 <= box_id & box_id <= 1024),  /* domain constraint */
  c_9: RANGE(1 <= length & length <= 1024),  /* domain constraint */
  c_10: RANGE(layer IN {"Metal", "Poly"});   /* domain constraint */
)

CLASS Port = (                             /* layout port */
 ATTRIBUTE
  port_id: INTEGER,                    /* port identifier */
  port_type: STRING,                   /* port type */
  direction: STRING,                   /* port orientation */
  location: Point;                     /* port location */
 CONSTRAINT
  c_11: RANGE(direction IN {"I", "O"}),          /* domain constraint */
  c_12: RANGE(1 <= port_id & port_id <= 300); /* domain constraint */
)

CLASS Wire = (                             /* layout wire */
 ATTRIBUTE
  wire_id: INTEGER,                  /* wire identifier */
```

```
 layer: STRING,                    /* layer on which the wire is placed */
 points: <Point>;                  /* the sequence of points defining the wire */
CONSTRAINT
 c_13: RANGE(layer IN  {"Metal", "Poly"}):     /* domain constraint */
)

/* entity interrelationships */

REPRESENTATION                     /* multiple representation abstraction */
 Layout (Cell [3], Gate [1]);      /* gate and its cell layout; the numbers in
                                   square brackets are association cardinalities */

/* database operations */

OPERATION
 BoxOverlap: Cell -> BOOLEAN,          /* semantic integrity constraint:
                                   BoxOverlap(c) is TRUE if some two boxes
                                   on the same layer in cell c overlap
                                   (constraint violated); is FALSE
                                   otherwise (constraint satisfied) */
 Area: Cell -> INTEGER,            /* area of a cell: Area(c) returns the area of cell c */
 Scale: Cell * INTEGER -> Cell,    /* geometric scaling of a cell: Scale(c, f)
                                   returns cell c scaled by a factor f */
 Overlap: Box * Box -> BOOLEAN;        /* checks if two boxes overlap:
                                   Overlap(b1, b2) is TRUE if boxes b1 and
                                   b2 overlap; is FALSE otherwise */
```

Figures (4.1) and (4.2) shows the resulting entity class composition and generalization/specialization hierarchies.

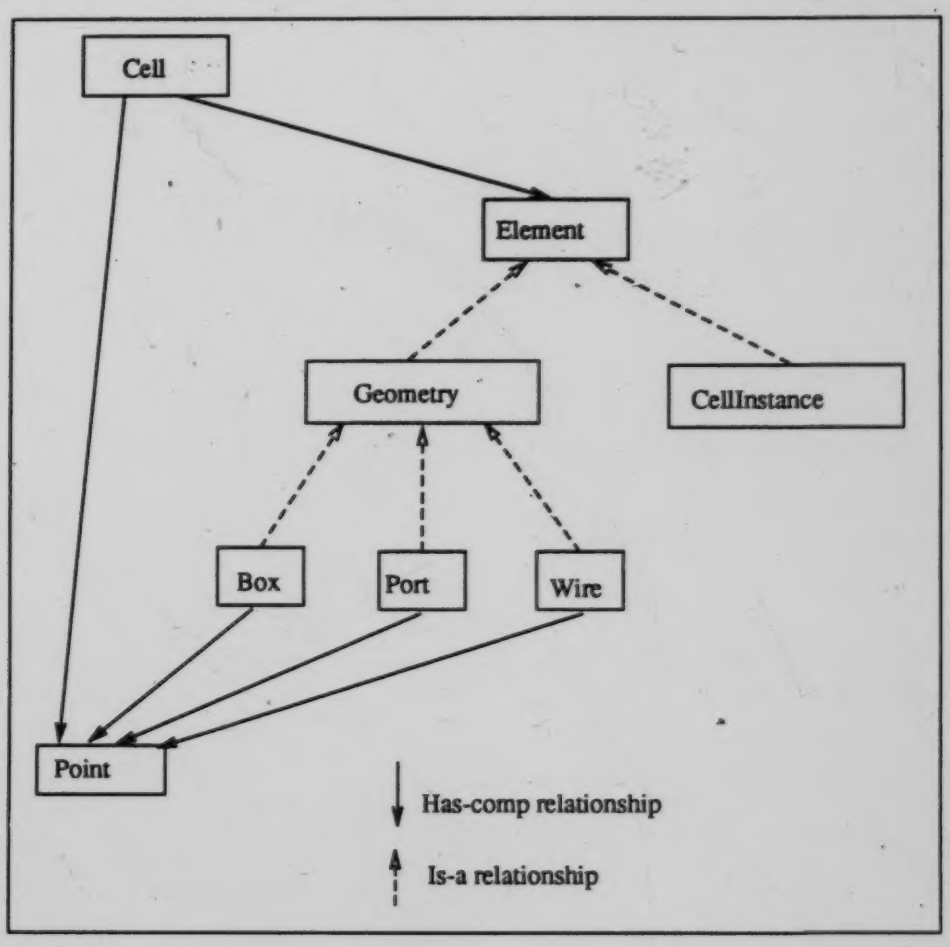

Figure 4.1 Entity Class Composition (Aggregation) Hierarchy

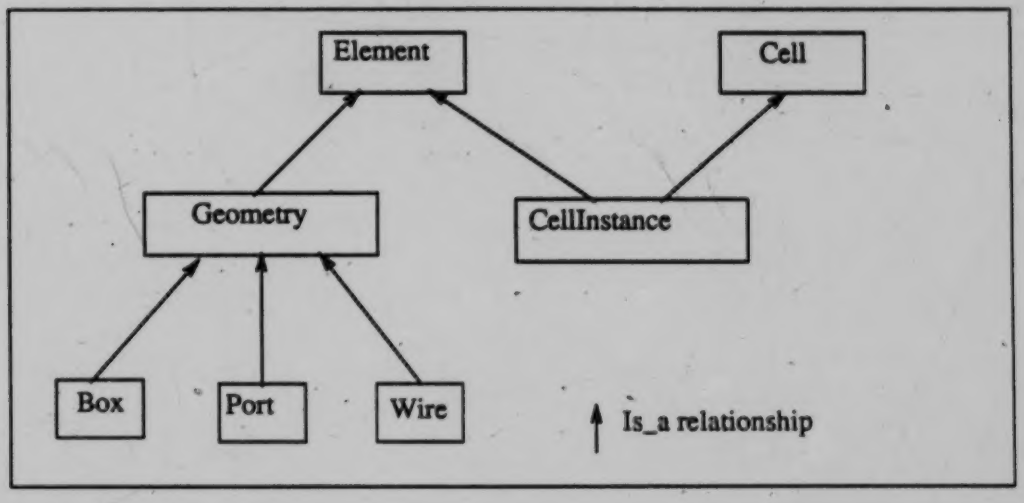

Figure 4.2 Entity Class Generalization/Specialization Hierarchy

4.7. Schema Definition Constraints

The *Is–a* relationship between entity classes has a many-to-many association cardinality. Thus a given entity class can have multiple generalizations and specializations. The *Is–a* relationship defines a "property inheritance" hierarchy by virtue of the induced class membership constraint. Thus, member entities of a subclass (having properties of that class) also belong (recursively) to all its superclasses as defined by the *Is–a* hierarchy (and so have the properties specified by the superclasses). Therefore, in addition to its own attribute and constraint properties, an entity subclass may redefine some of the inherited properties subject to the constraints described below.

4.7.1. Attribute Redefinition

The *attribute redefinition constraint* ensures that inherited attributes are not redefined inconsistently by a subclass.

Definition 4.15 If an attribute $A : T_1$ in entity superclass E_1 is redefined to be $A : T_2$ in entity subclass E_2, then the attribute redefinition constraint requires that:

$$\text{dom}(T_2) \subseteq \text{dom}(T_1)$$

The condition holds trivially when T_1 and T_2 are the same atomic types. For set (and sequence) types $S_1 = \{T_1\}$ (and $<T_1>$) and $S_2 = \{T_2\}$ (and $<T_2>$), we define:

$$\text{dom}(S_2) \subseteq \text{dom}(S_1) \text{ iff } \text{dom}(T_2) \subseteq \text{dom}(T_1)$$

where T_1 and T_2 may be entity classes or primitive data types. For entity classes E_1 and E_2, we define:

$$\text{dom}(E_2) \subseteq \text{dom}(E_1) \text{ iff } (E_2, E_1) \in Is\text{-}a^*$$

Furthermore, every multiply defined attribute has a unique most general entity class where it is defined.

4.7.2. Domain Constraint Redefinition

The *domain constraint redefinition constraint* ensures that inherited domain constraints are not redefined inconsistently by a subclass.

Definition 4.16: If a domain constraint $D : Pred_1$ defined by superclass E_1 is redefined to be $D : Pred_2$ in subclass E_2, then the domain constraint redefinition constraint requires that:

$(\forall e \in \text{Ext}(E_2))\ (\forall A \in \text{Attrib}(E_2) \cup_i \text{Attrib}(E_i)$ such that $(E_2, E_i) \in Is\text{-}a^*)$
$(Pred_2)[A\,(e)/A\,] \Rightarrow$
$(\forall A \in \text{Attrib}(E_1) \cup_j \text{Attrib}(E_j)$ such that $(E_1, E_j) \in Is\text{-}a^*)\ (Pred_1)[A\,(e)/A\,]$

which says that for every entity $e_2 \in \text{Ext}(E_2)$, if $(pred_2)[\text{A}(e_2)/\text{A}]$ (i.e. predicate $pred_2$ with $A(e)$ substituted for A where A is an attribute defined by E_2 or inherited from its

superclasses in the *Is-a* hierarchy) is true, then $(pred_1)[(A(e)/A]$ is also true. That is, for every entity in $Ext(E_2)$, if domain constraint D (in E_2) is satisfied then so is domain constraint D (in E_1). The index i in the union ranges over integers such that $(E_2, E_i) \in Is\text{-}a^*$ (the transitive closure of *Is-a* relation). In general, this may not be verifiable statically (at compile time) due to the undecidability of first-order predicate logic.

Example 4.10: The *CellInstance* class can redefine the domain constraint D inherited from the *Cell* class to be:

D: (1000 ≤ cell_id & cell_id ≤ 4000)

4.7.3. Key Redefinition

Definition 4.17: If a key K: $\{A_1, ..., A_k\}$ defined by a superclass is redefined in a subclass to be K: $\{A_1', ..., A_k'\}$, then the key redefinition constraint requires that:

$$\{A_1', ..., A_k'\} \subseteq \{A_1, ..., A_k\}$$

Example 4.12: Again, the *CellInstance* class may redefine the key constraint K inherited from the *Cell* class to be:

K = { *cell_id* }

4.7.4. Relationship Definition

The set of representation relationships defined in a valid database schema has to satisfy the *cross relationship constraint* defined as follows.

Definition 4.12: Given any two representation relationships:

$$R_i \subseteq \{i\} \times Ext(E_1) \times Ext(E_2)$$

$$R_j \subseteq \{j\} \times Ext(E_1') \times Ext(E_2')$$

we define the constraints:

$(E_1, E_1') \in HasComp^* \Rightarrow \neg((E_2', E_2) \in HasComp^*)$

$(E_2, E_2') \in HasComp^* \Rightarrow \neg((E_1', E_1) \in HasComp^*)$

for all pairs i and j.

The idea of this constraint is to ensure that the aggregate/component relationship between entities in one composition subhierarchy is not reversed by the corresponding representations in another composition subhierarchy.

Example 4.12: Consider the composition and representation relationships defined below:

$(Cell, Element), (Adder, Gate) \in HasComp^*$
$R_{layout_gate} \subseteq \{layout_gate\} \times Ext(Cell) \times Ext(Gate)$
$R_{layout_adder} \subseteq \{layout_adder\} \times Ext(Element) \times Ext(Adder)$

The above specifications are incorrect since the two representation relationships "intersect" as shown in Figure (4.3) below:

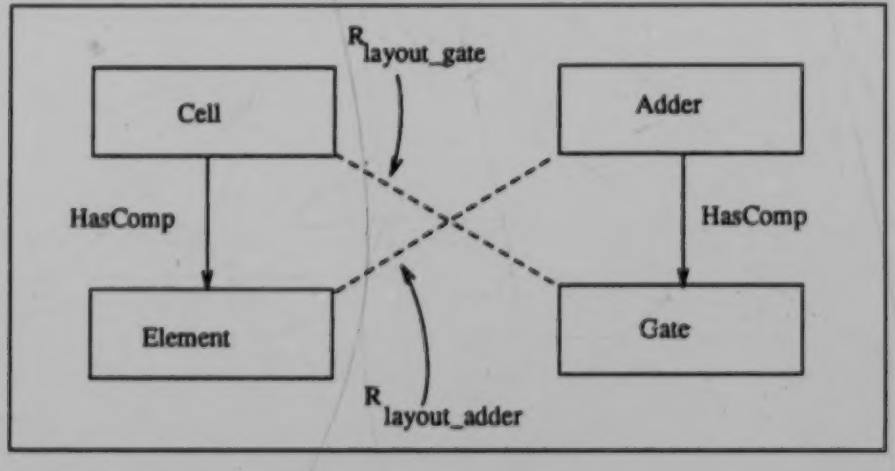

Figure 4.3 Cross Relationship Constraint

Analogous cross relationship constraints are defined for version and instance relationship sets.

4.8. Attribute Value Table

Definition 4.12: The set of all attribute values stored in the database at a particular time t defines the *attribute value table (AVT)*:

$$AVT = \{ (e, A, v) \mid e \in \text{Ext}(E) \text{ for some } E, A \in \text{Attrib}(E') \text{ for some } E' \text{ such that } (E, E') \in Is\text{-}a^*, \text{ and } A \text{ has type } T, v \in \text{dom}(T), \text{ and } \text{M}[A](e) = v \}$$

The attribute name (A) and entity (e) together determine the attribute unambiguously. The system looks up the definition of A starting from e's (most specific) class and chaining up the *Is-a* hierarchy. Attribute names can be multiply defined or reused provided they are in classes which do not share a common descendant, or if so related satisfy the attribute redefinition constraint.

4.9. Data Model Operations

A set of basic data model operations are defined. All other database operations of data manipulation, retrieval and update are defined in terms of these basic data model operations. We provide below an "operational definition" of the semantics of these operations.

(1) Accessing an entity class extension: Ext(E)

returns the extension of entity class E at the current time instant. For example, *Ext(Gate)* returns the set of *Gate* entities.

(2) Accessing an entity's class specification: Class(e)

returns the most specialized entity class of which the entity bound to the user-defined program variable e is a member. The most specialized class of an entity is always defined at entity creation time as the argument to the *CreateEnt* operation.

[step 1] return E such that $e \in \text{Ext}(E)$ &
$\neg (\exists E' \text{ such that } (E', E) \in Is\text{-}a^* \ \& \ e \in \text{Ext}(E'))$

For example, *Class(b)* returns (say) the entity class *Box* which indicates that b is a

Box entity.

(3) Creating an entity: $e \leftarrow$ CreateEnt(E)

creates a new database entity, inserts it into the extension of class E, and recursively, into the extensions of all its superclasses in the *Is-a* hierarchy, and binds variable e to the newly created entity:

[step 1] create a new entity e
[step 2] insert e into Ext(E)
[step 3] for all E' such that (E, E') $\in$ Is-a*, insert e into Ext(E')
[step 4] return e

Note that, after this operation, only the entity identifier is defined. This operation creates an entity which has all the properties (slots) of class E and all its superclasses, but no values are defined for any of these at this stage. Such an entity could be a called a "proto-entity". The attribute values have to be defined subsequently by invoking the data model operations. It is also possible to assign specified default attribute and relationship values at this time. For example, the operation $i \leftarrow$ *CreateEnt(CellInstance)* creates an entity, inserts it into the extensions of *CellInstance* and *Cell* and binds it to variable i.

(4) Deleting an entity: DeleteEnt(e)

deletes the entity bound to variable e from the database by removing it from the extensions of all entity classes of which it is a member:

[step 1] let $E \leftarrow$ Class(e)
[step 2] delete e from Ext(E)
[step 3] for all E' such that (E, E') $\in$ Is-a*, delete e from Ext(E')

The relationship instances involving the deleted entity are also deleted (although this need be done only when the relationship instance is referenced and not at entity deletion time). For example, *DeleteEnt(c)* would delete entity bound to variable c from the database.

(5) Assigning an entity attribute value: SET $A(e) = v$

assigns value v to attribute A of the entity e. The attribute A must be defined by class E or one of its superclasses in the *Is-a* hierarchy. If T is its associated value type, then $v \in dom(T)$. This model distinguishes between the concepts of "entity identity" and "attribute values"; entities can be created without any attribute values having been defined. Complex objects can now be built up incrementally by specifying appropriate attribute values. Thus the class definition only provides a "template" that defines what attributes are permissible and how they relate to other classes. Corresponding to each subset S of attributes with values defined there can be a *proto(E, S)* class. In contrast, the *mk-E* function constructs an object (or entity) of type (or class) E from predefined field (or attribute) values without using the notion of object (or entity) identity. Complex objects will then have to be constructed "bottom up" progressing from elementary objects to larger aggregates.

[step 1] let $E \leftarrow$ Class(e)
[step 2] if $\neg (A \in$ Attrib(E)),
then get E' such that $(E, E') \in$ Is-a* and $A \in$ Attrib(E') and A has type T;
[step 3] if $v \in$ dom(T),
then let $AVT \leftarrow AVT - \{ (e, A, *) \} \cup \{ (e, A, v) \}$
else return "value type error" message

where, in Step 3, * denotes a wild card character that matches any value. For example, *Set cell_type* (c) = "NMOS" would assign the value "NMOS" to the attribute *cell_type* of the entity c.

(6) Resetting an entity attribute value: RESET $A(e)$

removes the currently assigned value (if any) of attribute A of entity e.

[step 1] let $AVT \leftarrow AVT - \{(e, A, *)\}$

where * denotes a wild card character that matches any value. For example, *Reset cell_type* (c) would remove the assignment made in the example above.

(7) Accessing an entity's attribute value: $A(e)$

returns the value of attribute A of entity e; returns NULL if no value has been assigned yet or if a *Reset* operation has been executed.

[step 1] if tuple $(e, A, v) \in AVT$ then return v else return NULL

For example, *cell_type* (c) would return the value assigned to the *cell_type* attribute of entity c.

(8) Retrieving an entity set from the database: RETRIEVE ALL e: E WHERE (*var_def_seq* | $C(e)$)

retrieves all entities e in Ext(E) such that the condition $C(e)$ is satified; *var_def_seq* is an optional variable definition sequence that has the form e_1: E_1, ..., e_n: E_n where e_i are free entity variables and E_i are corresponding entity classes. This operation returns the set:

$\{ e \mid e \in \text{Ext}(E), e_i \in \text{Ext}(E_i) \text{ and } C(e) = \text{TRUE} \}$

For example, *RETRIEVE ALL c*: *Cell* WHERE ((*Layout*, c, g) $\in R_{Layout}$) would retrieve all *Cell* entities which are *Layout* representations of the *Gate* entity g.

(9) Selecting an entity from the database: SELECT e: E WHERE (*var_def_seq* | $C(e)$)

selects the entity e in Ext(E) such that the condition $C(e)$ is satified; *var_def_seq* is an optional free variable definition sequence that has the form e_1: E_1, ..., e_n: E_n where e_i are free entity variables and E_i are corresponding types. This operation returns the entity:

$e \mid e \in \text{Ext}(E), e_i \in \text{Ext}(E_i) \text{ and } C(e) = \text{TRUE}$

(10) ADT operations: The composite types of Set and Sequence each provide the usual set of associated abstract data type operators (as described in Chapter 7).

Entity Relationship operations: The following operations are defined for manipulating the representation relationships.

(11) Creating a representation relationship instance: CreateRep(i, e_1, e_2)

creates an i relationship instance between the representation entity e_1 and the abstract entity e_2:

[step 1] insert the tuple (i, e_1, e_2) into R_i.

(12) Deleting a representation relationship instance: DeleteRep(i, e_1, e_2)

deletes the relationship instance between entities e_1 and e_2 defined by representation relation R_i:

[step 1] delete the tuple (i, e_1, e_2) from R_i.

(13) Retrieving representations of a given abstract entity: GetRep(i, e_2)

returns the set of representation entities for a given abstract entity e_2 as defined by the representation relation R_i:

[step 1] return { e_1 | (i, e_1, e_2) ∈ R_i }

(14) Retrieving the abstract entity corresponding to given representation: GetAbs(i, e_1)

returns the abstract entity corresponding to the given representation entity (e_1) as defined by the representation relation R_i:

[step 1] return entity e_2 such that (i, e_1, e_2) ∈ R_i

The *GetAbs* operation defines an equivalence relation on the representation entities as follows:

Equiv(e_1, e_1') iff GetAbs(i, e_1) = GetAbs(i, e_1')

Analogous operations are defined for version and instance relationships.

Example 4.13: Consider a database of persons, employees and violin players. A partial schema for such a database is defined by the class definitions in Table (4.2) which results in the *Is-a (subclass of)* hierarchy shown Figure (4.4): *Person, Employee* and *Violin* player entities can now be created by means of the *CreateEnt* data model operation as

Class	Attribute	Constraint
Person	SS# Name Address	Subclass: Employee, Violin-player
Employee	Title Department	
Violin-player	Skill-level	

Table 4.2 Partial Database Schema

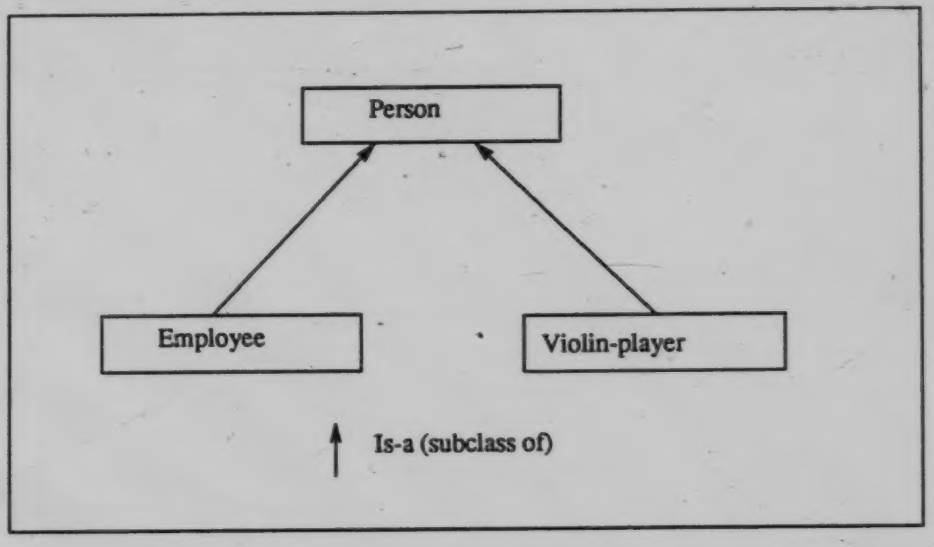

Figure 4.4 *Is-a* Hierarchy of Classes

follows:

```
p = CreateEnt(Person)          /* creates a new entity p which is inserted
                               into the extension of class Person
                               (denoted "Ext(Person)" or "Persons") */
e = CreateEnt(Employee)            /* creates a new entity e which is inserted into
                               the extensions of class "Employee" (denoted
                               "Employees") and (by using the "Is-a"
                               hierarchy) class "Person" ( denoted "Persons") */
v = CreateEnt(Violin-player)       /* creates a new entity v which is inserted into
                               the extensions of classes "Violin-player"
                               ("Violin-players") and "Person" ("Persons") */
```

But now the only way to specify that person *p* is also employee *e* and violin player *v* is to set up a user-defined relationship between entities *p*, *e* and *v*. Alternatively, a new entity class *Violin-player-employee* can be defined (statically) as shown in Table (4.3).

Class	Attribute	Constraint
Person	SS# Name Address	Subclass: Employee, Violin-player, Violin-player-employee
Employee	Title Department	
Violin-player	Skill-level	
Violin-player-employee		

Table 4.3 Partial Database Schema

The *Is-a (subclass of)* hierarchy is now modified as shown in Figure (4.5). Now the operation:

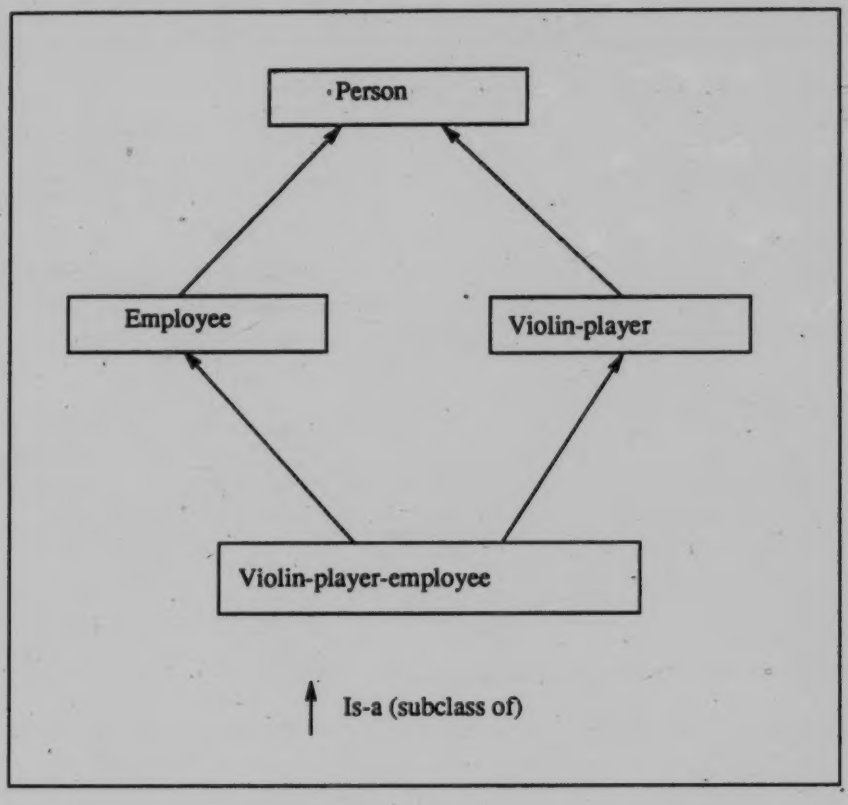

Figure 4.5 *Is-a* Hierarchy of Classes

p = CreateEnt(Violin-player-employee)

creates a new entity *p* and inserts into extensions of classes *Violin-player-employee, Employee, Violin-player* and *Person*. Thus *p* will represent (simultaneously) a person, an employee and a violin player. This brings up the issue of defining new classes by (set) intersection of the specified entity class extensions. The class *Violin-player-employee* could be defined as the intersection of the classes *Employee* and *Violin-player*. The DBMS can then automatically insert *p* into the extension of *Violin-player-employee*. But this solution involves modifying the *Is-a* hierarchy dynamically or there may not always

be a most specific class for every entity.

Relaxing the 'most specialized class' constraint on entities

Consider a new data model operation:

INSERT ENTITY *entity* INTO *entity-class-list*

which inserts an existing entity *entity* into the extension of classes in the list *entity class list* and all their superclasses. This permits entities to be specialized dynamically by inserting existing entities into new entity class extensions and defining appropriate new property values for them. But now an entity may no longer have a single most specialized class.

Example 4.13a: To define person *p* simultaneously as an employee and a violin player, the following solution could be used:

```
p = CreateEnt(Person);
INSERT ENTITY p INTO Employee, Violin-player;
```

Now entity *p* represents a person, an employee, and a violin player, but has no most specific class. In this case the operation *Class(e)* returns the set of minimal entity classes of which *e* is a member. We can define more general operations than those above which maintain the concept of a class as corresponding to the properties an object has: subclasses have more properties in all cases. The effect obtained would be the same as if the quasi-order of classes were completed to an intersection semilattice, where all intersections of classes, if non-empty, are present. The details of this will not be given here.

4.10. Structural Integrity Constraints

The following structural integrity constraints are defined:

(1) *Entity Identity:* Every database entity has a globally unique *identifier* which is independent of its attribute values and relationship instances in which it may participate. This avoids the need to define keys and associated proliferation of key uniqueness, subset and foreign key constraints of the relational model [CoM84]. Whenever a new entity is to be created, the system supplies a globally unique identifier. Entity identifiers are system-defined and keys are user-defined, but both can be used in retrieval operations. Entity identifiers are made visible by providing a system-defined attribute (say) *EntID*. Entity identifiers can be simple integers or structured with fields corresponding to server name, entity name, time of creation, owner etc. If key attributes are defined, then it is possible for the system to ensure that the key attribute values uniquely identify an entity; of course there may be several entities with the same values for some subset of the key.

(2) *Referential Integrity:* In our data model referential integrity [Cod81] is enforced through abstract domains defined by the entity class extensions. Any entity defining an attribute value of another entity, or a relationship instance, is verified by the system to be a member of the appropriate entity class extension. If the referenced entity is not found in its entity class extension, the reference is void and the attribute is assigned a NULL value, and the relationship instance is deleted from the database. That is, the deletion of a referenced entity nullifies the reference. Formally, referential integrity is defined in terms of "valid" values. A value v is valid if:

(a) if v is a basic data type value (including NULL) then it is a valid.

(b) else if $v = e$ (an entity) and $e \in \text{Ext}(E)$ for some E then v is valid.

(c) else if $v = \{v_1, ..., v_N\}$ or $v = \langle v_1, ..., v_N \rangle$ and $v_1, ..., v_N$ are valid values, then v

is valid.

Referential integrity in the database can now be enforced by ensuring that:

(a) all entity attribute values are valid:

for all tuples $(e, A, v) \in AVT$, $e \in \text{Ext}(E)$ for some E, $A \in \text{Attrib}(E)$ and, v is a valid value

(b) all relationship instances are valid:

for all tuples (i, e_1, e_2) in R_i (and V_i and I_i):
$e_1 \in \text{Ext}(E_1)$ for some entity class E_1 and $e_2 \in \text{Ext}(E_2)$ for some entity class E_2
and the relationship functionality constraint is satisfied:
if f_i = <m, n>, then:
cardinality($\{e_1 \mid (i, e_1, e_2) \in R_i$ (and V_i and I_i) for a given $e_2\}$) $\leq$ m
and,
cardinality($\{e_2 \mid (i, e_1, e_2) \in R_i$ (and V_i and I_i) for a given $e_1\}$) $\leq$ n

(3) *Entity Typing:* All entities are "typed" by the entity classes to which they belong. The generalization/specialization abstraction hierarchy of entity classes models both the "type inheritance" and "inclusion polymorphism" mechanisms [CaW85]. Thus entities can belong to multiple classes corresponding to different levels of abstraction at which they can be viewed.

4.11. Database State

The database state σ is defined collectively by the extensions of all the entity classes in the schema, the attribute value table, and the representation, version and instance relationships:

$$\sigma = (\langle \text{Ext}(E_i) \mid i=1, .., n \rangle, AVT, \bigcup_j R_j, \bigcup_k V_k, \bigcup_l I_l \bigcup_R G_R)$$

4.12. Database Operations

The semantics of the design database are partly defined by a set of user-definable, application-specific data manipulation and retrieval operations which may require the full power of general programming languages (Turing computability). These operations may be used to (1) manipulate, query and retrieve information from the database, (2) compute derived values, or (3) check complex semantic integrity and consistency constraints specified procedurally. The database operations are specified as functions which map values from a domain defined by the cross product of domains corresponding to entity classes, primitive data types or set and sequence data types, to a similarly defined codomain. Thus a database operation O specifies the function:

$$O: D_1 \times ... \times D_m \rightarrow C_1 \times ... \times C_n$$

where D_i and C_j are value domains defined by the atomic types or set or sequence types. Operation names can be overloaded (analogous to operators in programming languages) provided they can be distinguished by their input domains D_i.

The concept of *operations* in our model differs fundamentally from that in systems like Smalltalk: operations are not part of a class but have an independent status. An operation takes data values or entities belonging to a certain class (and potentially to several different subclasses) as input arguments and returns data values or entities as result arguments. Hence, there is no system-defined inheritance of operations. However, an operation that takes an entity in a subclass can use an operation that takes an entity in a superclass, because the entity in the subclass is also in the superclass and has all the superclass properties.

Example 4.14: Consider the entity classes and operations listed below. The class *Window* has subclass *Fancy-window*. The operation *Insert* inserts a specified text string into the specified coordinate location in a specified window. Similarly the operations *Rotate-*

text, Insert-menu-items and *Get-speech* act on a specified fancy window. These operations are implemented as database language* procedures and functions with appropriate parameters as shown below.

Class	Attribute	Constraint
Window		Subclass: Fancy-window
Fancy-window		

Table 4.4 Partial Database Schema

Operation	Implementation
Insert	PROCEDURE Insert(IN w: Window; x, y: INTEGER; text: STRING)
Rotate-text	PROCEDURE Rotate-text(IN f: Fancy-window; angle: INTEGER)
Insert-menu-items	PROCEDURE Insert-menu-items(IN f: Fancy-window; items: {Item}) TYPE Item :: x: INTEGER; y: INTEGER; text: STRING;
Get-speech	FUNCTION Get-speech(f: Fancy-window; time-interval: INTEGER): Sound

The *Insert* procedure takes input values *w* (window), *x, y* (position) and *text* (text to be inserted) and performs the desired operation. Similarly for the other operations. The procedure *Insert-menu-items* can use the *Insert* operation (which acts on windows) as follows:

* described in detail in Chapter 5

```
PROCEDURE Insert-menu-items(IN f: Fancy-window; items: {Item})
BEGIN
 ....
 VAR i: Item; x, y: INTEGER; text: STRING;
 /* insert items into fancy window using the "Insert" operation */
 FOR i = Elems(items) DO
  x = x(i); y = y(i); text = text(i);
  Insert(f, x, y, text);
 ENDFOR
 ....
END
```

The invocation of *Insert* above will work correctly because the fancy window *f* has all the properties of a window. The *Get-speech* function can be defined in such a way that it does not change any properties of *f* (both as a window and as a fancy window) but associates a sound with it.

These database operations can be implemented in a variety of ways.

(1) Relation tables may explicitly store the graph of the function defined by the operation. Data manipulation and retrieval is then defined by a set of relational algebraic (table manipulation) operations.

(2) A high-level, query/data manipulation language statement (or group of statements) may define the result in terms of other, previously defined operations or the basic data model operations defined in Section 4.9.

(3) A user-written, high-level integrated "database language" procedure or function may process input arguments to compute the desired result. We shall elaborate on such a database language in the next chapter.

4.12.1. Data Manipulation, Querying and Retrieval

Complex data manipulation, querying and retrieval operations can be written as database language routines that use the basic data model operations in appropriate statements.

Example 4.15: Consider the following partial schema shown in Table (4.5) below:

Class	Attribute	Value Type
Resistor	resistance	REAL
Amplifier	resistors	{Resistor}
Cell	amplifier	{Amplifier}
	cell-id	INTEGER

Table 4.5 Partial Database Schema

and the update "Increment the resistance values of all resistors in amplifiers in the cell with *cell-id* = 19 by 50%". The following database language program fragment executes the specified update:

```
VAR x-set: {Resistor};
  x: Resistor;

/* Step 1 [ Retrieve the desired resistors ] */

 x-set = RETRIEVE ALL r: Resistor WHERE (a: Amplifier, c: Cell | (cell-id(c) =
     19) & (a IN amplifier(c)) & (r IN resistors(a)));

/* Step 2 [ Change the resistances ] */

 FOR x = Elems(x-set) DO
  SET resistance(x) = resistance(x) * 1.5;
 ENDFOR
```

4.12.2. Derived Attributes

These can be specified and implemented as database operations which compute the derived values.

Example 4.16: The derived attribute *area* of a cell entity can be modeled as a database operation with the mapping specification:

area: Cell → Integer

and can be implemented by the database language function which simply computes the area of the rectangle defined by the lower left and upper right corners of the bounding box of the cell as shown below.

```
FUNCTION area(c: Cell): INTEGER
BEGIN
 RETURN ((x(ur_corner(c)) - x(ll_corner(c))) *  /* length of cell multiplied by */
     (y(ur_corner(c)) - y(ll_corner(c))));      /* width of cell */
END
```

Here, x and y are attributes of a *Point* class which specify the Cartesian coordinates of a *Point* entity.

4.12.3. Complex Semantic Integrity and Consistency Constraints

A design database is characterized by complex semantic integrity constraints. In a VLSI design database, for example, *composition, consistency and correctness constraints* may represent "design rules," mutual consistency of different representations of the same design entity, or update protocols that define how one part of the database is to be modified in response to an update to another part. These constraints are specified in terms of *constraint checking functions* which can be implemented as database operations whose execution can be triggered by the D-DBMS in response to user requests or database updates.

Example 4.17: The cell composition constraint that two boxes may not overlap on a given layer can be specified as a constraint checking function *BoxOverlap* with the mapping specification:

BoxOverlap: Cell → Boolean

and can be implemented as the recursively defined database operation shown below:

```
/* recursive function to evaluate the semantic integrity constraint BoxOverlap:
  returns TRUE if boxes on the same layer of the cell overlap (constraint
  not satisfied) or FALSE otherwise */

FUNCTION BoxOverlap (c: Cell): BOOLEAN   /* c is the formal parameter that
                                            represents the cell entity for
                                            which the integrity constraint is
                                            being checked */

 VAR e1, e2: Element;                    /* declare local entity variables */

BEGIN
 FOR e1 = Elems(component(c)) DO         /* iterate over the set of components
                                            in cell c; elems is the iterator
                                            that successively yields elements
                                            in the set value component(c) */
  IF (e1 IN Ext(CellInstance))           /* if the element is a CellInstance
                                            (Ext is the data model operation
                                            that returns the extension of a
                                            class) */
  THEN IF (BoxOverlap(e1))               /* recursively invoke BoxOverlap */
     THEN RETURN(TRUE);
    ENDIF
  ELSE FOR e2 = Elems(component(c)) DO    /* iterate over the set of elements in cell c */
     IF (e2 IN  Ext(CellInstance))        /* if the element is a CellInstance */
      THEN IF (BoxOverlap(e2))            /* recursively invoke BoxOverlap */
          THEN RETURN(TRUE);
         ENDIF
      ELSE IF ((e1 IN Ext(Box)) AND (e2 IN Ext(Box)) AND (layer(e1) = layer(e2))
          /* elements are both boxes and are on the same layer */
          THEN IF Overlap(e1, e2)         /* check two boxes overlap */
              THEN RETURN(TRUE);
             ENDIF
         ENDIF
     ENDIF
    ENDFOR
 ENDIF
ENDFOR
```

```
  RETURN(FALSE);          /* no two boxes on the same layer overlap */
END
```

Here, *component* and *layer* are attributes defined in the entity class *Cell*; *Overlap* is another database operation that checks if two specified boxes are overlapping. Essentially, the algorithm implemented by the above function does the following for each pair of *Element* entities composing the given *Cell* entity: (1) if either one of them represents a *CellInstance* entity, invoke *BoxOverlap* recursively; (2) if both of them represent *Box* entities on the same layer, then invoke *Overlap* to check if they overlap.

The function *BoxOverlap* can be triggered by an integrity subsystem whenever a box is made a component of a cell. If the function returns TRUE the constraint is violated and the update is disallowed; otherwise the constraint is satisfied and the update allowed.

4.13. Partially Defined Design Objects in CAD Databases

Here we consider a method for database operations to handle partially defined design objects which are represented in the design database by entities with values specified for a subset of their applicable attributes. Consider, for example, a database operation *DbOp* with the mapping specification:

$$\text{DbOp: } D_1 \times ... \times D_m \rightarrow D_1' \times ... \times D_n'$$

where D_i $(1 \le i \le m)$ and D_j' $(1 \le j \le n)$ are value domains: entity classes, basic data types, or set or sequence types. Let the operation be implemented by a database language procedure *DbProc* with the calling interface:

$$\text{PROCEDURE DbProc(IN } o_1: D_1; ...; o_m: D_m; \text{ OUT } o_1': D_1'; ..., o_n': D_n')$$

where o_i $(1 \le i \le m)$ and o_j' $(1 \le j \le n)$ are the formal input and output parameters respectively.

Let the operation *DbOp* be invoked by the procedure call:

DbProc(a_1, ..., a_m, a_1', ..., a_n')

where a_i ($1 \le i \le m$) and a_j' ($1 \le j \le n$) are the actual input and output arguments corresponding to the formal parameters. These arguments may be entities, atomic values (of basic data types), sets or sequences. Except in the first case (entities), the arguments are "fully defined", and the procedure can execute correctly with a well-defined result. In the case of entity arguments, however, correct procedure execution cannot be statically guaranteed, as entities may represent only "partially defined" design objects: that is, only some of their attributes may have values specified for them. The database operation invocation protocol requires only the entity identifiers (procedure *call by reference* or *pointer semantics*). A correct execution of the procedure then requires dynamic (run-time) *compatibility* and *definedness* checks to be made. These checks could be specified by predicates defined within the scope of the procedure that implements the database operation.

Formally, we define a compatibility predicate as follows. If entity e_i is an argument corresponding to the formal parameter o_i with "type" entity class C_i in procedure *DbProc*, then we define the predicate *compatible* as follows.

(a) if every entity has a single most specialized class:

$$\text{compatible}(e_i, o_i) \Delta (\text{Class}(e_i), C_i) \in Is\text{-}a^*) \text{ OR } (\text{Class}(e_i) = C_i)$$

(b) if entities do not have a single most specialized class:

$$\text{compatible}(e_i, o_i) \Delta (\exists C \in \text{Class}(e_i) \cdot ((C, C_i) \in Is\text{-}a^* \text{ OR } C = C_i) \,\&\, e_i \in \text{Ext}(C))$$

where the symbol Δ is to read as "is defined as" and $Is\text{-}a^*$ is the transitive closure of the $Is\text{-}a$ (subclass) relation on entity classes. That is, an entity e_i belonging to C_i or any subclass of C_i is compatible with the formal parameter o_i. Since the system automatically inserts an entity belonging to a certain class recursively into all its superclasses as defined by the *Is-a* relationship, the compatibility predicate is implemented simply by checking that e_i is a member of the extension of class C_i ($e_i \in \text{Ext}(C_i)$). Thus, the

subclass/superclass class hierarchy provides universal polymorphism by inclusion which allows an operation to be applied to entities belonging to different classes.

Formally, we define a definedness predicate as follows. Define the *DefState* function as follows:

$$\text{DefState}(o_i) \Delta \{ A_1, ..., A_p \}$$

where $A_1, ..., A_p$ are attributes in class C_i (both explicit and inherited) such that if the entity argument corresponding to the formal parameter o_i has appropriate non-NULL values specified for these attributes, "correct" execution of the procedure is guaranteed. Thus, the *DefState* function applied to a formal parameter computes a "sufficiency set" of attributes. The user may provide the system with a "minimal" sufficiency set of attributes, or alternatively, provide "hints" on how to compute it. If no such data is supplied, the system computes the sufficiency set (which may not be minimal) by a static analysis of the procedure. For an entity argument e_i corresponding to the formal parameter o_i, define the *DefState* function as:

$$\text{DefState}(e_i) \Delta \{ A_1', ..., A_q' \}$$

where $A_1', ..., A_q'$ are attributes of e_i that have non-NULL values defined. Multiple attribute definitions with the same attribute name can be disambiguated by considering the class C_i (the class of the formal parameter o_i) and the entity e_i (the argument) and its classes. Then, we define the *definedness* predicate as follows:

$$\text{defined}(e_i, o_i) \Delta \{ A_1, ..., A_p \} \subseteq \{ A_1', ..., A_q' \}$$
$$<=> \text{DefState}(o_i) \subseteq \text{DefState}(e_i)$$

The procedure *DbProc* performs the required compatibility and definedness checks using the above predicates. If either of these two predicates (*compatible* and *defined*) evaluates to FALSE, then the procedure is not executed, but an *undefined* exception is returned. If the procedure *DbProc* calls other procedures with entity arguments, then these called procedures, in turn, perform the compatibility and definedness checks with respect to their

parameters and arguments, returning the undefined exception if the predicates evaluate to FALSE. Thus, the undefined exception can be propagated up the procedure call hierarchy to the top (user) level where a suitable error message can be printed. By tracing the execution down through the procedure call hierarchy, it is also possible to indicate where the undefined exception is raised and which undefined attribute of which entity caused the exception.

4.14. Summary

In this chapter, we have presented and formally specified a conceptual-level, semantic data model that provides the design database concepts of entities, entity classes, attributes, constraints, entity interrelationships, basic data model operations, and structural integrity constraints. A unique feature of the model is that it provides a framework for handling design objects that are partially defined and whose definition can be built up incrementally. This semantic data model may be described as an *entity-oriented* model, because database entities correspond directly to application domain objects. Several "object-oriented" database systems have appeared in recent literature. We feel that the essential criterion for evaluating object-oriented databases should be whether they provide the notion of a database object (or entity) representing an application domain object. While object-oriented languages like Smalltalk, C++, Ada etc. provide features that simplify an implementation of the entity-oriented semantic data model, it is by no means necessary to work within the framework (and hence limitations) of these languages. As we show in Chapter 5, none of these languages addresses the entire range of issues of importance in implementing an entity-oriented semantic data model, and hence a new database language which combines features of traditional object-oriented languages with database-specific requirements is needed. This integrated database language provides an interface between the semantic data model and the application domain operations, and

allows clear and concise specification of complex semantic integrity and consistency constraints, and application-specific operations. We present the design of such an integrated database language in the next chapter.

Chapter 5

AN OBJECT-ORIENTED DATABASE LANGUAGE FOR A CAD DBMS

5.1. Introduction

A database language provides users with an interface to the database system. Traditionally this database language interface has been defined at three different levels. The *data definition language* (DDL) provides the database administrator a means of defining the database schema in terms of the basic structuring constructs of the data model and the associated auxiliary information about access privileges, accounting information etc. The *data manipulation language* (DML) provides application programmers with a means of accessing and manipulating the database through database system calls embedded in their high-level application programs. Typical data manipulation operations may include data retrieval, data update, and report generation. The data and control (database state) information communication between the application program and the DBMS is achieved through main memory buffers shared between the two address spaces. The *query language* (QL) enables the casual, non-programmer user to retrieve data and information through ad-hoc queries. Typically, query languages are high-level, non-procedural, command-like languages, which can often be stylized natural language commands.

The design of a database language and its capabilities depend, of course, on the underlying data model supported by the DBMS. Engineering CAD applications are highly interactive, iterative, incremental, and involve complex operations and semantic integrity and consistency constraints. It is therefore necessary to provide designers with a single, unified interface to the design database that combines data definition, manipulation and query capabilities in an *integrated database language*. One aspect of current research on design database systems has addressed this problem of providing a suitable database language interface. However, as we shall describe below, none of these efforts

have been completely successful for two primary reasons:

(1) The database system is based on a classical data model, which as shown previously, is unsuited for design databases; and,

(2) The database language is seen as an interface that bridges the semantic gap between the application program/user and the data model supported by the DBMS rather than as an integrated system that has the required semantics built into it.

The detailed analysis below should therefore convince the reader that a new integrated database language tailored to capture the abstractions of the design database semantic data model is an essential component of a CAD database system architecture. In Section 5.9 we describe the design of our proposal for such a database language.

5.2. Database Language Design Issues

The design of a database language centres around the following issues:

- Data Abstraction
- Polymorphism and Property Inheritance
- Data Persistence
- Type System Implementation
- System Architecture
- Partially Defined Objects
- Design Transaction Control
- Set-oriented Retrieval and Processing

We shall consider some of these in some detail below.

5.3. Data Abstraction

Abstraction is a powerful concept that enables us to deal with complexity by separating the specification, implementation and usage aspects of a software system. The basic objective of *data abstraction* is to hide the details of data structuring from users and to restrict them to using a set of well-defined operations through which they may access and manipulate the data structure. This enables not only modularization of software by encapsulating data structures and operations, but also enforces security by preventing incorrect manipulations of data. Data abstraction in software systems is achieved through the notion of data types. A *data type*, in the context of programming languages, is defined as a set of values (or data objects) - its *domain* - and a set of legal *operations* on these values. Data types help to ensure "authentication" (only legal operations are applied to values), and "protection" (illegal modifications of the representation are prevented) [Gut80]. A data type implementation has two components - its *representation* (which defines the structure of the values of the type) and its *operators* (which define how the operations are implemented). A comprehensive treatment of the evolution of abstraction in programming languages is found in [Gua78].

Let us consider the type systems of typical programming and database languages proposed in recent literature. Aldat [Mer78], Pascal/R [ScM80] and Modula/R [Rei84] all provide a *relation* data type which enables them to directly manipulate database relations using relational algebra or relational calculus. C++ [Str84] provides *int* (integer), *float* (real), and *char* (character) primitive data types, *pointer*, *array*, *structure* and *union* composite data types, and the *class* construct which enables definition of ADTs. *E* [RiC87] extends C++ by providing a *dbclass* construct that defines a persistent class and whose instances represent persistent database values. CLU [LiG86] provides the usual primitive data types, the *array*, *sequence*, *variant*, *oneof* (union), *record*, and *struct*

(tuple) composite data type constructors, and the *cluster* construct which enables definition of ADTs. Galileo [Alb84, ACO85] provides the usual primitive data types, tuple, sequence, union, modifiable value (*var*) and function composite data type constructors, and a *class* construct for ADT definition. Poly [Mat85] is a strongly typed language that features a data type system in which types consist of sets of *objects* (which may be data values or procedures) and a *type mark*; types are "first class" values (which can be created and manipulated at run-time like any other values) and type specifications are "textual" and are separately compilable units (like the modules of some other languages). Poly types include the usual primitive data types, the *record, struct* and *union* composite data type constructors (and corresponding operations), and the *type* construct that enables definition of ADTs.

5.4. Polymorphism and Property Inheritance

Cardelli and Wegner [CaW85] provide a taxonomy of forms of polymorphism in programming languages. *Universal polymorphism* involves generic procedures (or functions) that can process values from an infinite number of types all having a common structure. This form of polymorphism is further classified into *parametric polymorphism* in which the specific type to which the argument value belongs is implicitly or explicitly supplied through a type parameter, and *inclusion polymorphism* which involves viewing values as belonging to different types with overlapping domains. An example of a generic procedure which exhibits parametric polymorphism is a *length* function that returns an integer value given a list of elements of arbitrary type. Inclusion polymorphism relies on a hierarchy of types defined by the subtype/supertype relationship between types and the fact that a value of a subtype can be used in contexts where a value of the supertype is expected. Such a hierarchy defines an *extensional constraint* (values belonging to a subtype also belong to a supertype; that is, domain(*subtype*) $\subseteq$ domain(*supertype*)), and an

intensional constraint (operators of a supertype are inherited by the subtype) [Alb84, CaW85]. For example, in the case where type *car* is defined as a subtype of the type *vehicle*, inclusion polymorphism implies that the domain of *car* is a subset of the domain of *vehicle*, and hence, every *car* instance is also a *vehicle* instance, and every operation that can be applied to *vehicle* instances can also be applied to *car* instances.

Ad-hoc polymorphism, on the other hand, involves a small set of "monomorphic procedures" that are implicitly invoked to process values from a corresponding small set of types that do not have any apparent common structure. Ad-hoc polymorphism by *overloading* involves using the same name (or syntactic notation) to denote different procedures; the selection of a particular procedure depends on the context and the type of argument value supplied. For example, the addition operator (+) is normally overloaded to denote addition of different combinations of integer and floating point values. Ad-hoc polymorphism by *coercion* involves an implicit or explicit conversion of a given argument value to an appropriate value of the desired type expected by the procedure. For example, an integer value can be converted to a floating point value before an addition operator is applied.

Let us now consider the data typing and polymorphic features of some high-level languages. Algol-68 which allows run-time coercions like a two dimensional array with only one row to be converted to a vector, and pure Lisp with untyped values and plenty of run-time coercions, illustrate clearly the danger of erroneous results due to coercions when they arise from programming errors and not by design. This has led to a trend towards strongly typed languages that prohibit automatic coercions and require all coercions to be explicitly specified by the programmer. Simula's [DMN70] class hierarchy provides inclusion polymorphism with subclass objects being permitted whenever a superclass object is expected. Lisp Flavors [WeM81] extends this inheritance hierarchy

by allowing multiple immediate superclasses, and Amber [Car84] extends the notion of inheritance to higher-order functions. ML [Mil84] introduced the notion of parametric polymorphism with type variables that can be instantiated to different types in different contexts. ML's type system also enables the most general type that fits a given specification to be infered, so that it is possible, for example, to write a generic sort procedure that works for elements of any type with an ordering relation. Other languages which provide parametric polymorphism include CLU [LiG86] and Poly [Mat85]. The concrete types of Galileo [ACO85] are organized into a property inheritance hierarchy based on a system-defined *is* (subtype) relationship between the types. Classes (and derived classes) are also organized into a user-defined subset hierarchy that supports inclusion polymorphism. Taxis [MBW80] is an entity-based database specification language that provides a tree-structured hierarchy of entity classes, transaction classes and value classes defined by an *IsA* relationship that specifies both an extensional constraint (subclasses are contained in superclasses) and a structural constraint (properties are inherited and may be redefined so that the original and the new definitions, in turn, satisfy the IsA relationship). Again this organization supports inclusion polymorphism. IRIS [Fis87] provides a *type hierarchy* defined by the subset/superset relationship between "object types" that supports multiple (type) inheritance and inclusion polymorphism.

In the case of design databases, polymorphic values arise naturally as a consequence of the generalization/specialization hierarchy of entity classes (entities are members of multiple classes); and, generic procedures are required to model operations that can be applied to entities belonging to different entity classes. Consider for example, the VLSI circuit layout database scheme described in the example given earlier. The entity class hierarchy defined by the specifications makes a *Box* entity also a member of the

Geometry class, and a *CellInstance* entity also a member of the *Cell* class. Similarly, the derived value function *area* is polymorphic as it can be applied to both *CellInstance* and *Cell* entities.

5.5. Data Persistence

Traditionally, persistence of data has been provided by file and database management systems with application programs making service calls to these subsystems to move data between the program's address space in main memory and the persistent store. Data persistence in programming languages involves one of the following techniques [BuA86].

(1) The "all-or-nothing" approach, exemplified by languages like Lisp or Prolog, enables everything in a program state to be saved on secondary storage ("snapshot") and recovered for later execution. This has the disadvantages that the saved state is not sharable among different users and there is no distinction made between temporary and persistent values.

(2) *Replicating persistence* involves moving values in and out of secondary storage by means of explicit input-output requests. Although languages like Pascal and APL provide this kind of persistence, there are severe limitations on the type of values that can be replicated.

(3) In *intrinsic persistence* every program value is, by definition, persistent and is replicated on secondary storage thus eliminating, at least conceptually, any distinction between primary and secondary storage.

The problem with the conventional approach to data persistence is the loss of data abstraction capability provided by the language's data type system as a result of the mapping between main memory and the persistent store. Data persistence and data typing

should be viewed as orthogonal properties. That is, all values have a persistence attribute independent of their data types. Furthermore, a spectrum of data persistence can be visualized. In order of increasing data persistence, different "degrees" of persistence can be associated with:

(1) Intermediate values in expression evaluation.

(2) Data values of local variables (in the procedure's activation record stored on the stack).

(3) Data values of global variables (stored on the heap).

(4) Data values retained between executions of a program.

(5) Data values retained between executions of versions of a program.

(6) Data values that outlive versions of a program.

In contrast to Pascal or APL, replicating persistence in the database context requires that the type information also be replicated along with the values, so that when the database is recovered and referenced subsequently the appropriate type information is available. This essentially requires making all database values have a "persistent data type" and a "global scope" within the application program.

PS-Algol [ABC83] is an extension of S-Algol that transparently provides data persistence so as to enable database applications to manipulate database and program values uniformly. Its data type system consists of scalar types (*integer, real,* etc.), vector type (**T* which defines vectors of elements of type *T*), and a pointer (to a structure) type (*persistent identifier* (PID)). Persistent values are stored and manipulated on the heap. The criterion for persistence of values on the heap is "reachability" of a value starting from a persistent "root value" defined explicitly by the user. The specification, access and manipulation of persistent values is achieved through a "functional interface" consisting of a

set of system-supplied procedures (or functions) callable from the application program. Whenever a database value is moved on to the heap (as a result of a call to the *open.database* procedure) a pointer to the main memory structure representing the database value is returned (to the calling routine). The pointer (which has the type PID) can be used as a component of other structures to build up main memory representations of complex database values. The initial PID value representing the entire database is, of course, returned by the *open.database* call. The program variables bound to these PIDs are global, and on transaction commit or program termination, the updated data values referenced by these variables are written out to the database.

The database language Amber [Car84] provides replicating persistence through the *dynamic* type, which is a most general type whose domain contains all database values of interest; a dynamic value can be "coerced" to a value of appropriate type using the coerce instruction provided in the language. The mapping between database values and the (file) structure on secondary storage is established through *extern* and *intern* instructions.

E [RiC87] provides a *dbclass* (database class) mechanism which is basically a persistent class whose objects (instances) can reside either on secondary storage or in main memory, and can be explicitly moved between these areas by the programmer. The members of a dbclass must all be dbclass objects. Dbclasses are user-definable and may be parameterized with data type and function parameters. The system provides a set of built-in dbclasses including *dbint, dbfloat, dbchar, dbvoid* etc. Pointers to dbclass objects can be declared just as with normal classes and data types, but operations on such pointers are restricted to assignment and dereferencing. Dbclass objects can only be declared "locally," and are allocated on the heap. E also provides the concept of a *file* which is a built-in, persistent dbclass that stores an unordered sequence of dbclass objects of one specific dbclass. Operations to create and destroy a file instance and to store and

retrieve objects are provided by the file dbclass.

In the database context, therefore, persistence is essentially achieved through two mechanisms: "schema", which provides a name space, and "persistent values" (of persistent types) which are automatically saved on and retrieved from secondary storage.

5.6. Type System Implementation

A *type system* implements a language's data type concepts (built-in types and type constructors) by providing a set of *representations* (data structures), *operators* (procedures and functions), and *auxiliary procedures* that provide *polymorphism* (type hierarchy, type parameterization, coercion and overloading), and *storage management* (allocation, deallocation, garbage collection etc.). If the language includes an ADT feature, then the type system also provides *access controls* to regulate access to the type's representation instances.

A type's representation has implications with respect to certain properties of the type and how its instance structures are allocated and manipulated in main memory. For example, a mutable set type may have a linked list representation; now the set denoted by a variable of the type can be modified efficiently by the type's operators of insert, delete etc. A *mutable type* thus implies that the size of the instance structure can change during program execution and hence heap storage is required for values of the type. In contrast an *immutable type's* instance structures are fixed in size, and hence can be created in a program's stack area. The other associated concept of relevance to a type's implementation is that of value vs. pointer semantics. *Value semantics* for a type implies that variables of the type can be visualized as "containing" (abstract) values (or objects) of the type, and operations like assignment, initialization, argument and result passing, and equality testing involve distinct objects which may represent the same or different

abstract values. With mutable types this means that a separate instance structure is required for each variable of the type and operations of assignment and argument and result passing duplicate the instance structure while the equality test compares two distinct instance structures for equality. With immutable types, however, the instance structures can safely be shared among multiple variables of the type that have the same abstract value. Equality testing then requires checking that the variables are bound to the same instance structure.

Example 5.1: Consider a mutable *Stack* ADT that provides value semantics. A set of stack declarations, operations and corresponding semantics can now be written as:

```
[1] VAR x, y: Stack;/* declare stacks x and y */
[2]    i: INTEGER;
[3] x = Push(x, 4);   /* assign to x (left value) a stack obtained by pushing 4
                         on to the stack contained in x (right value) */
[4] x = Pop(x);       /* assign to x (left value) a stack obtained by popping the
                         stack contained in x (right value) */
[5] i = Top(x);       /* assign to i the top of the stack x */
[6] y = x;            /* assign to y a stack equal to that contained in x */
[7] IF (x = y)        /* predicate is TRUE if both x and y contain "equal" stacks */
  THEN ..
  ELSE ..
```

The statements in lines 3 and 4 are executed by simply modifying the instance structure bound to variable x. The assignment in line 6 duplicates and binds to variable y the instance structure bound to variable x. The test in line 7 compares distinct instance structures bound to variables x and y.

Consider now an immutable *Complex-number* ADT that provides value semantics. A Complex number declaration, the assignment and complex conjugate operation and corresponding semantics can be written as:

```
[1] VAR a, b, c: Complex-number;   /* declare complex number a, b and c */
[2] b = a;                         /* assign to b the complex number a */
[3] c = Complex-conjugate(a);      /* assign to c a complex number obtained by
                                      taking the complex conjugate of the complex
                                      number a */
```

The assignment in line 2 simply binds variable *b* to the instance structure bound to variable *a*, while the assignment in line 3 creates and binds to variable *c* a new instance structure which represents the complex conjugate of the complex number represented by the instance structure bound to variable *a*.

With *pointer semantics* on the other hand, variables of the type are just pointers to (abstract) values of the type. Operations like assignment, argument and result passing, and equality testing involve pointers to the values. Instance structures can now be shared among multiple variables of the type that have the same value and operations of assignment, argument and result passing just create additional pointers while the equality test compares pointers to instance structures.

Example 5.2: Consider the mutable *Stack* ADT that now provides pointer semantics. A set of stack declarations, operations and corresponding semantics can now be written as follows:

```
[1] VAR x, y: Stack;     /* declare stack pointers x and y */
[2]    i: INTEGER;
[3] Push(x, 4);          /* push 4 on to the stack pointed to by x */
[4] Pop(x);              /* pop the stack pointed to by x */
[5] i = Top(x);          /* assign to i the top of the stack pointed to by x */
[6] y = x;               /* assign to y the stack pointer x */
[7] IF (x = y)           /* predicate is TRUE if both x and y point to the same stack */
    THEN ..
    ELSE ..
```

The operations in lines 3 and 4 simply update the instance structure bound to the variable x. The assignment in line 6 binds variable y to the instance structure bound to variable x. The test in line 7 compares the bindings of x and y.

Thus, the two aspects of mutable vs. immutable type and pointer vs. value semantics interact strongly, and together define the properties of a type and how the type system implements them.

The other aspect of a type system implementation deals with polymorphism through type parameterization, type hierarchies, coercion (of values) and overloading (of operator symbols). Polymorphic procedures and functions are procedures or functions defined with implicit or explicit type parameters along with specifications of operations that are to be provided by these types. The type system can then check that argument types satisfy such constraints. An example of such a type system is that of CLU [LiG86]. In CLU, a polymorphic function *p* could be defined as:

```
p = proc [t: type] (x: t) returns(t)
  where t has equal:proctype(t, t) returns(bool)
```

where *t* is a type parameter with the constraint that it provide an operation *equal* which tests for equality of values of the type, and *x* is formal argument. *p* is now a function that takes an argument value of type *t* and returns a result value also of type *t*. The type hierarchy, which provides inclusion polymorphism, is defined in terms of a

subtype/supertype relation between types. The type system can then replace a subtype by its supertypes to infer type consistency, and hence permit values of a subtype to be used when a value of a supertype is expected. For example, Galileo's [Alb84] type system defines a "concrete type" hierarchy in terms of an **is** (subtype) relation as follows:

(1) For any type t, (t **is** t).

(2) For tuple types r and s of the form $(a_1: t_1, ..., a_n: t_n)$, ($r$ **is** s) iff (a) the set of attributes of r contains the set of attributes of s, and (b) if r' and s' are the types of a common attribute, then (r' **is** s').

(3) For variant types r and s of the form $<a_1: t_1, ..., a_n: t_n>$, ($r$ **is** s) iff (a) the set of tags of r is contained in the set of tags of s, and (b) if r' and s' are the types of a common tag, then (r' **is** s').

(4) For sequence types r and s of the form "**seq** t" with elements of types r' and s', (r **is** s) iff (r' **is** s').

(5) For modifiable types r and s of the form "**var** t", (r **is** s) iff the associated types are the same.

(6) For function types $(r \rightarrow s)$ and $(r' \rightarrow s')$, $(r \rightarrow s)$ **is** $(r' \rightarrow s')$ iff (r' **is** r) and (s **is** s').

In implementing polymorphism by coercion, the type system provides appropriate conversion procedures to convert between instance structures (values) of different types. For example, an expression (4.6 + 3) is compiled into the expression (4.6 + float(3)) where *float* is a system library function that converts an integer argument into a real number and returns the real number. Thus, the language C++ [Str84] provides the following type coercions:

char → short → int → long; float → double; int → double

In implementing polymorphism by overloading, the type system replaces the overloaded operator or function symbol by the appropriate procedure. For example, in C++, an overloaded *date* constructor may be specified as:

```
date(int, int, int);    /* day, month and year */
date(char*);            /* date as a string value */
date(int);              /* day, assume current month and year */
date();                 /* today, the default date */
```

with a different procedure corresponding to each of these constructors. The type system will substitute the appropriate procedure depending on the type and number of arguments in any invocation of the constructor. Stroustrup [Str84] gives an example of a *string* data type (defined in terms of C++ class (ADT) facilities) which implements string values as pointers to string objects, provides value semantics for operators of the type by creating a copy of the string value whenever required (initialization, assignment and argument passing in a "call by value" procedure call), and provides all the required storage management.

5.7. System Architecture

Typically, there are three approaches to implementing the database language interface:

(1) Special database language constructs are embedded in a host programming language [Sch80]. These database statements are preprocessed by a preprocessor to generate host language calls to a *DBMS Interface* (DBI) module. The resulting database program can now be compiled by the host language compiler and linked to the DBI module at load time if it is precompiled with appropriate procedure linkage conventions; it can also be linked to the DBI module at run time if a *Monitor* module can mediate the communication between the application process and an independently executing DBI process. Examples of this architecture include EQUEL [Sto74],

Modula/R [Rei84], Pascal/R [ScM80], PLAIN [Ker81]. Alternatively, the database constructs can be encoded in terms of an intermediate data structure representation while the pure host language application code is compiled. This allows the system to package and optimize a set of related database operations in terms of "database transactions" which can take advantage of the underlying database structure for efficient execution. Adaplex [SFL83] is an example of this approach to implementing a DBMS.

(2) A special-purpose, "integrated" database and programming language is provided to the application programmer. This enables us to eliminate the "impedance mismatch" between the two components. Now, database applications written in the *integrated database language* can be compiled into machine code by a corresponding compiler and executed. An example of this architecture is TAXIS [MBW80].

(3) The database language program can be translated into an intermediate encoded representation (data structure) which can be interpreted at run time by a *System Interpreter* component of the DBMS. Examples of this architecture include query languages like QUEL [Sto74] and GEM [Zan83], and integrated database languages like GLIDE2 [EaT79], EFDM [KuA86], Galileo [ACO85], Poly [Mat85] and Amber [Car84].

5.8. Integrated Database Language for a D-DBMS: A Conceptual Basis

We shall proceed by listing the semantic data model abstractions to be captured by the integrated database language, considering how each of these can be provided by the language system, and discussing the related capabilities and features of several languages proposed in the literature.

• *Entity*

Entities are the basic abstractions of the design database. From the point of view of the database language, entities can be modeled by combining the concepts of the *abstract data type* (ADT) and the *object paradigm*. An *Entity* ADT would provide data and procedural abstraction by encapsulating the representation (data structures) and the operations (procedures) applicable to entities. With the development and subsequent popularization of Smalltalk [GoR83], there has been a proliferation of "object-oriented" languages, databases and systems. Consequently, object-orientation has become a buzzword with different, and often imprecise, meanings. We define below precisely what we mean by the object paradigm and show how this is useful in the context of our database language. The object paradigm provides three basic concepts:

(1) *Objects* which can be visualized as instances of ADTs that encapsulate data structures and operations on these structures (data abstraction). Thus, any language which provides an ADT feature can, in a limited sense, be considered an object-oriented language.

(2) *Inheritance* of representations and operations by "subtypes" from "supertypes". This can be provided by organizing the set of ADTs in the property inheritance type hierarchy defined by the supertype/subtype relationship, so that subtypes inherit the data structures and operations from supertypes, and add new data structures and operations in addition to refining or redefining the inherited data structures and operations. Again, the existing object-oriented languages exhibit a range of inheritance mechanisms.

(3) *Object identity* provides each object with a unique identity so that objects can be assigned to variables, shared among different variables, procedures, programs and even users, used as building blocks of more complex structures which may, in turn, be instances of other ADTs, and passed as arguments to and returned as results from

procedures. Several existing languages with both ADTs and inheritance, however, do not provide this notion of object identity, and therefore do not qualify as fully object-oriented.

Wegner [Weg87] provides a taxonomy of the various forms of the object paradigm and analyses how these concepts have been manifested in several different programming languages.

Translated into the context of our proposed database language, therefore, the object paradigm provides each instance of the *Entity* ADT with a unique identifier thus enabling entities to be treated as "first class" values: entities can now be assigned to variables, used as building blocks of more complex structures, shared among multiple structures, passed as arguments to and returned as results from database operations, and as we shall see later, can be made persistent. Extended languages like Pascal/R or Modula/R have neither an *Entity* ADT nor an object paradigm. Enhanced languages like Modula-2 and C++ do not have a built in *Entity* ADT; C++ for example, provides a *class* construct that enables a user to define an *Entity* ADT, but does not support the object paradigm which implies a fundamental limitation with respect to its usefulness as a database language.

• *Complex Objects*

Complex objects, which are sets or sequences of entities and basic data type values, are another basic abstraction of the design database data model. To implement this at the database language level, we need parameterized *set* and *sequence* composite types, again in combination with the object paradigm which enables us to manipulate these complex objects as atomic entities. Again, most of the proposed database languages do not provide these composite types although it is possible for the user to define these types using the ADT facility provided in languages like C++, Galileo (class), Clu (cluster), Poly (type), and Ada (package).

• *Classification*

Our design database data model provides the classification abstraction in terms of the *class* construct. A *class extension* defines a set of entities that share a collection of common properties defined by the *class intension*. These shared properties are defined in terms of *attributes* which map entities to values in specified *value domains*, and *constraints* which are predicates on the attribute values of an entity. The notion of the class can also be captured by an ADT, but we choose to keep the notions of ADT and class orthogonal so that we can combine the two concepts when necessary, and yet be able to distinguish between them. This design provides us with modeling expressiveness and flexibility which would otherwise be lost.

Again, several of the proposed languages in the literature provide the concept of ADTs (although the construct that defines an ADT is often called *class*), but not the notion of class as a set. A user defined ADT that implements a class as a set in such a language, however, does not preserve the distinction between the two concepts. Languages like Galileo, Taxis and Amber provide the notion of a class as set, which, however, is not orthogonal to the concept of an ADT, and hence the two cannot be combined.

• *Generalization/Specialization Hierarchy*

This concept enables design entities to be viewed at different levels of abstraction by allowing them to belong to different classes and letting them inherit the attributes and constraints defined for these classes. These entity classes are organized in a *property inheritance hierarchy* defined by the *superclass/subclass* relationship between the classes. Subclasses inherit properties from superclasses thus capturing the generalization/specialization abstraction. As the inheritance hierarchy can, in general, be a directed graph (and not just a tree) our model supports multiple inheritance. Inheritance

constraints define how properties can be inherited from multiple superclasses without conflicts or ambiguities. Since the database language implements the entity class as a set, the class generalization/specialization hierarchy defined by the superclass/subclass relationship can be implemented as a set hierarchy defined by the set inclusion relation. A second useful consequence of the generalization/specialization abstraction is that it enables *polymorphic operations* in which operations with entity parameters belonging to a certain class can also operate on entities belonging to their subclasses.

Examples of languages with class hierarchies include C++ and Smalltalk (single inheritance), and Taxis and Adaplex (multiple inheritance). Other languages like Amber, Galileo and Poly provide polymorphism through *type hierarchies* defined in terms of supertype/subtype relationship between parameterized composite types like tuple, union, set and sequence etc. Yet other languages like Clu and Ada provide *parametric polymorphism* which involves passing types as arguments to procedures.

• *Entity Interrelationships*

Our design database data model also provides a set of predefined, "typed", parameterized relationships that capture the semantics of the design entity interrelationships like *Repr-of, Version-of* and *Instance-of*. The parameters of such relationship specifications are the "type" of the relationship (i.e. representation, version or instance), the entity classes involved, the association cardinality and user-defined procedures (protocols) to be invoked whenever the basic relationship operations (create, delete, or retrieve) are applied. The database language can provide these interrelationship abstractions as parameterized ADTs. Again, none of the proposed languages provide the entity interrelationship abstraction which puts the burden of setting up and maintaining such relationships and associated semantics on the user.

• *Persistence*

Design database entities are persistent, which implies that they persist beyond an execution instance of the database program. Conventional approaches to persistence have been described earlier Section 5.2. Our integrated database language can provide persistence by making the entity class, the *Entity* ADT, and the database schema persistent; that is, all entity classes, entities, and values reachable starting from these entities are automatically saved on disk by the system upon program/session termination. This also eliminates any distinction between main memory and disk at the conceptual level and the user is presented with a one-level persistent store. There is, therefore, no user-level I/O instruction required to move data between the program address space and the database on disk. The system can, of course, choose to optimize the amount of I/O required in storing and restoring persistent entities by using several techniques including demand-driven I/O, caching, and restoring only updated entities etc. Among the proposed languages only Poly provides selective persistence although Smalltalk, Galileo and Taxis all provide intrinsic persistence in which all of the program address space is automatically saved by the system.

• *Complex Database Operations, Semantic Integrity and Consistency Constraints*

A design database is characterized by complex operations, semantic integrity and consistency constraints which, in general, require the power of a Turing machine (computable functions) for their specification and implementation. The integrated database language, therefore, has to be a full-fledged, general-purpose programming language. The trend in current proposals for integrated database language has thus been toward increasing computing power, and in fact all of the languages we have discussed above are Turing complete.

In the next section, we combine the concepts discussed above in the framework of a

general-purpose, Pascal-like language to come up with the design of our *Integrated Database Language* (IDL).

5.9. IDL System Design

5.9.1. Values

The basic elements of IDL are *values* which are data objects (instances of primitive or abstract data types) created and manipulated in an IDL program. A value is a basic data type value, an entity, or a tuple, set or sequence of basic data type values or entities. Values exist independently of procedure/function invocations. Main memory space for new values is allocated automatically by the system. The system also reclaims the memory space allocated to a value when it is no longer "accessible". Values may be divided into two categories. *Mutable values* are those which have a modifiable "state" which can be changed without changing the "identity" of the value (time-varying behaviour). Entities, sets and sequences are mutable values. *Immutable values* are those values which have no modifiable state. Integers, reals, Booleans, strings and tuples are immutable values.

5.9.2. Type System

IDL provides the basic data types of *Integer, Real, String* and *Boolean*; the composite data types *Tuple, Set* and *Sequence*; and, abstract data type *Entity*. Type specifications appear in variable declarations. The basic data types and the tuple type are immutable types while the composite data types and the abstract data type *Entity* are mutable types. Each type has an associated *representation.*

Integer Type

The values of this basic data type constitute a subrange (defined by the length and format

of the bit representation in the machine used to encode integers) of integers in mathematics. It is an immutable type. The operations of the type are + (sum), - (difference), * (multiplication), / (division), ** (exponentiation), *mod* (modulo), and comparison operations of = (equal), < (less than), <= (less than or equal), > (greater than), and >= (greater than or equal).

Real Type

The values of this basic data type are a subset (defined by the length and format of the bit representation in the machine used to encode reals) of the reals in mathematics. It is an immutable type. The operations of the type are + (sum), - (difference), * (multiplication), / (division), ** (exponentiation), *float* (convert an integer value to a real value), *round* (round a real value to an integer value), *trunc* (truncate a real value to an integer value), and comparison operations of = (equal), < (less than), <= (less than or equal), > (greater than), and >= (greater than or equal).

String Type

The values of this basic data type are sequences of ASCII characters. It is an immutable type. The operations of the type are *substr* (substring), *length*, || (concatenate), *indexc* (position of a specified character in the string), *indexs* (position of a specified substring in the string), *chars* (iterator over the characters in the string), and comparison operations of = (equal), < (less than), <= (less than or equal), > (greater than), and >= (greater than or equal).

Boolean Type

This basic data type has the values TRUE and FALSE. It is an immutable type. The Boolean operations are & (AND), | (OR), ~ (NOT), and the comparison operation = (equal).

Sequence Type

Values of this composite type are variable length sequences of elements which can be basic data type values, tuples, or entities. A *Sequence* type is specified by applying the sequence type constructor (<...>) to a specific element type parameter as shown in the examples below:

```
<Point>      /* sequence of Point entities */
<INTEGER>    /* sequence of integers */
<REAL>       /* sequence of reals */
```

The *Sequence* type is a mutable type, and its values may be manipulated by the type's operators described in Chapter 7.

Set Type

Values of this composite type are sets of elements which can be basic data type values, tuples or entities. A *Set* type is specified by applying the set type constructor to a specific element type parameter as shown in the examples below:

```
{Point}      /* set of Point entities */
{INTEGER}    /* set of integers */
{REAL}       /* set of reals */
```

The *Set* type is a mutable type, and set values may be manipulated by the type's operators described in Chapter 7.

Tuple Type

Values of this composite type are tuples consisting of sets of attribute-value pairs. A tuple type is specified by applying the tuple constructor to specified attribute-domain pairs. The example below shows the specification of a tuple type *Event* with attributes *signal, value* and *time*:

```
Event ::    signal: STRING,
            value: BOOLEAN,
            time: INTEGER;
```

The tuple type is an immutable type and tuple values may be manipulated by the type's operators described in Chapter 7.

Entity Type

This is an abstract data type. The values of this type represent database entities. A variable whose type is an entity class defines an *Entity* type object (database entity). As shown in the example below, the variable *c* represents an entity belonging to class *Cell.*

```
CLASS Cell = ( ... )      /* define entity class Cell; the ellipsis denotes
                             omitted property specifications */
VAR
 c: Cell;                 /* c is a Cell entity implemented as a value of the
                             abstract type Entity */
```

The associated *Is-a* class hierarchy defines an inclusion polymorphism for values of the *Entity* type: the entities are members of the extension of their entity classes and also, recursively, their superclasses in the *Is-a* hierarchy. The operations of the abstract type *Entity* are the basic data model operations defined in Section 4.9.

Type Representations

The basic data types and *Set*, *Sequence* and *Tuple* composite data types have the representation defined by the structure *Value_Rep*:

```
struct Value_Rep {
 int            valueID;              /* value ID (system generated) */
 char           struct_type[6];       /* structure type {Int, Real, Bool,
                                         String, Set, Seq, Tuple, Entity} */
 union ValueType_Rep value;           /* actual value */
};
```

The *Entity* abstract data type has the representation defined by the structure *Entity*:

```
struct Entity {                       /* persistent database entity */
 int entityID;                        /* internal entity identifier */
 struct Class_Def *class;             /* pointer to most specific entity class */
};
```

The operations of the *Entity* type are the basic data model operations defined in Section 4.9.

5.9.3. Variables

Variables are fundamental "things" in the IDL universe. A variable has a *type* and may have a *value* that it currently denotes. A variable is said to be uninitialized if it does not denote any value.

(1) Variables can be introduced by variable declarations as shown in the examples below:

```
VAR
 i: INTEGER;              /* integer value */
 r: REAL;                 /* real value */
 s: STRING;               /* string value */
 p: Point;                /* a Point entity */
 point_set: {Point};      /* a  set of Point entities */
 point_seq: <Point>;      /* a  sequence of Point entities */
```

(2) A value may be assigned to a variable.

(3) A variable may be used as an expression; the value of such an expression is then the value denoted by the variable at that time.

5.9.4. Expressions

An *expression* evaluates to a value in the IDL universe. The simplest form of expressions are the *value constants* (integers, reals, literals etc.) and *variables*. More complex expressions are built up by combining simpler expressions using type operators, function invocations and/or value returning data model operations. Most type operators are specified by an infix or prefix syntax; others are specified as invocations of user-defined functions. When an expression involving infix or prefix operators is not fully parenthesized, the proper grouping of subexpressions may be ambiguous. To resolve this ambiguity, operators are assigned precedence values, and applied in order of decreasing precedence. The operator precedences are as shown in Table (5.1).

Operator Precedence Table	
Operator	Precedence
|(OR)	0
&(AND)	1
<, <=, =, >, >=	2
+, -	3
*, /	4
**	5
~(NOT)	6
-(minus)	7

Table 5.1 IDL Operator Precedence Table

Operators with the same precedence are applied from left to right.

5.9.5. Assignment

Assignment of a value to a variable is a basic operation in IDL and has the form:

```
x = E;   /* x is a variable and E is an expression */
```

The effect of the assignment above depends on the type of x. If the type of x is a basic data type or a *Set* or *Sequence* type, then a new value (equal to that obtained by evaluating *E*) is constructed, and *x* is set to denote that value (*value semantics*). If x has the abstract type *Entity* or the composite type *Tuple* then it is set to denote the same value obtained by evaluating *E* (*pointer semantics*). The assignment semantics are illustrated graphically below. Figure (5.4) shows variables *x, y, a, b, u* and *v* with their associated values. Figure (5.5) shows the state after the following assignments:

```
y = x;
b = a;
v = u;
```

Some examples of assignments are:

```
i = 2;                                  /* i has type INTEGER */
r = 3.1412;                             /* r has type REAL */
flag = TRUE;                            /* flag has type BOOLEAN */
st = "This is a string";                /* st has type STRING */
int_set = { 1, 2, 3 };                  /* int_set has type {INTEGER} */
real_seq = < 32.1, 45.678, 091.2 >;     /* real_seq has type <REAL> */
ent_set = { c1, c2, c3 };               /* c1, c2, c3 have type Cell and
                                           ent_set has type {Cell} */
e = mk-Event("x", TRUE, 10);            /* e has type Tuple */
```

In IDL variables hold pointers (left values) to the values they denote. The mutable types

are *Set, Sequence* and *Entity*. Updates to values denoted by variables of these types are defined in terms of the appropriate type operators which abstract from the low-level concerns of manipulating left values. For example, a sequence value can be defined and manipulated as follows:

```
VAR s: <INTEGER>;     /* declare a sequence variable */
  n: INTEGER;

s = emptySeq();       /* create an empty sequence and bind it to s */
s = append(s, 2);     /* append value 2 to the sequence */
s = insert(s, 1, 10); /* insert value 10 in the 1st position in the sequence */
n = select(s, 1);     /* select the 1st element in the sequence */
n = length(s);        /* compute the length of the sequence */
 ...
```

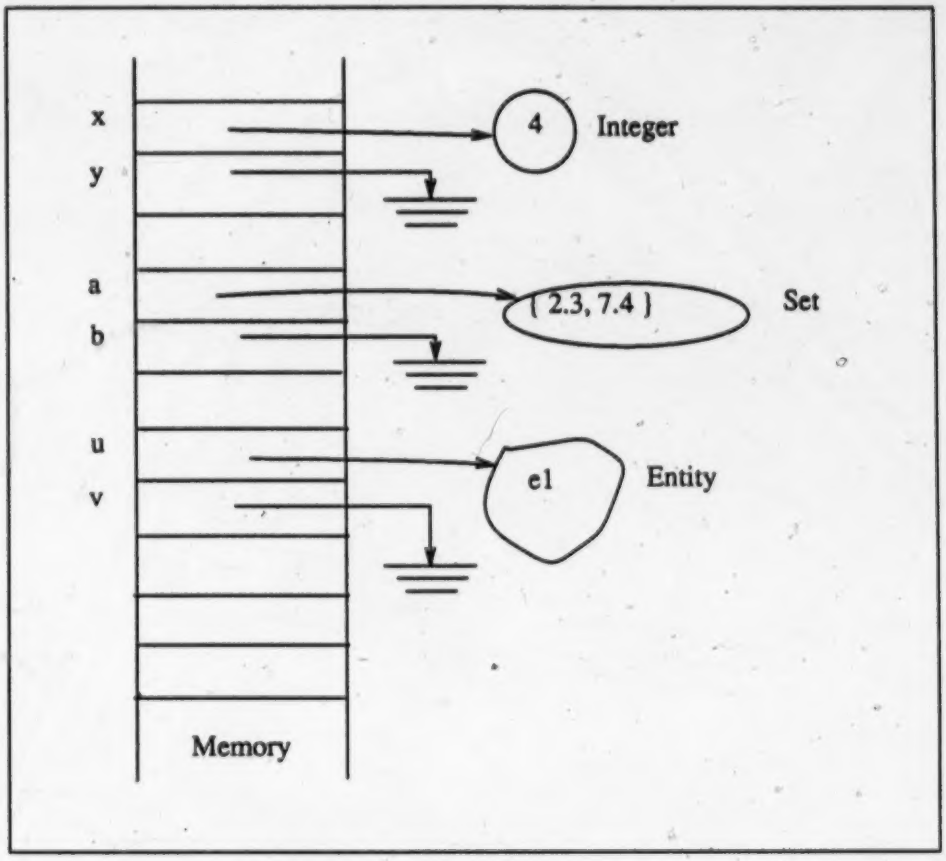

Figure 5.1 Variables and Values in IDL

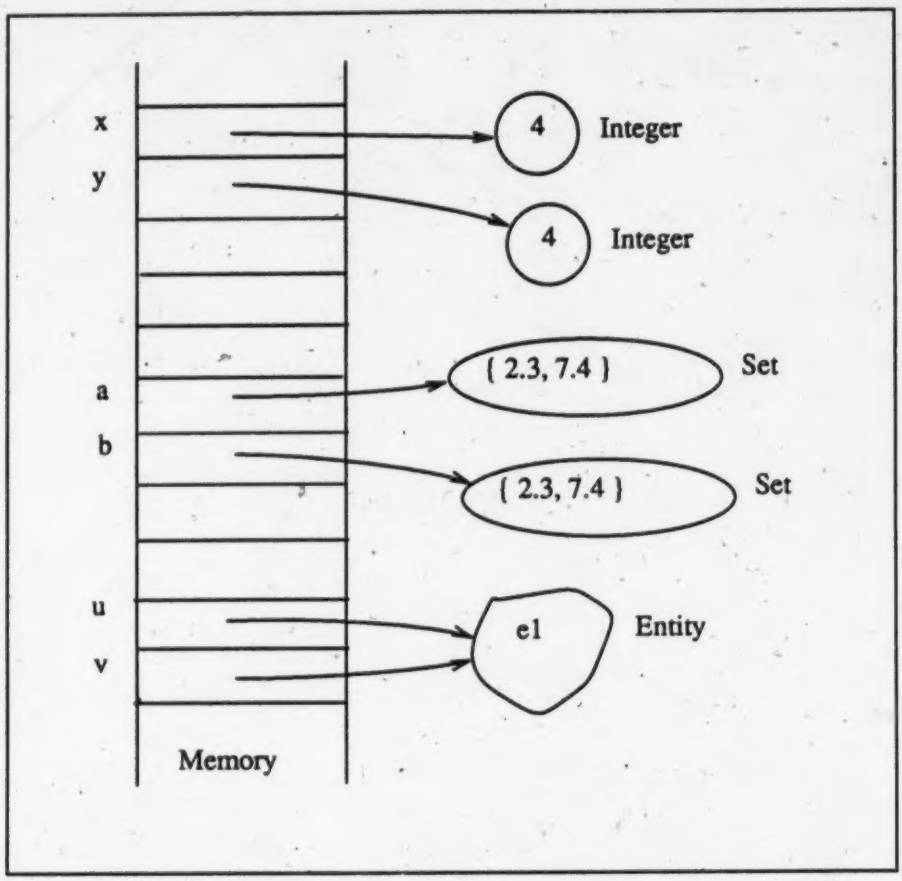

Figure 5.2 Variables and Values in IDL: The State after specified Assignments

5.9.6. Procedures and Functions

Procedures and functions encapsulate data declarations and control structures within a local environment. Both procedures and functions perform an action on zero or more argument values. A procedure terminates without returning any value while a function terminates by returning a single value of a specified type. Procedures and functions are generically called *routines*. A routine declaration has a *header* and a *body*. A procedure

header specifies a *procedure name* and a sequence of input and output *formal parameters* and has the form:

PROCEDURE name (IN p_1: T_1; ....; p_m: T_m; OUT p_{m+1}: T_{m+1}; ....; p_n: T_n);

Here *name* specifies the name of the procedure, p_1 through p_m are formal input parameters with corresponding types T_1 through T_n, and p_{m+1} through p_n are formal output parameters with corresponding types T_{m+1} through T_n. When the procedure is invoked the input parameters are initialized by assignment of corresponding argument values. When procedure execution terminates, the output arguments are initialized by assignment of corresponding formal parameter values computed by the procedure.

A function header specifies a function *name*, a sequence of input *formal parameters*, and a result value *type*, and has the form:

FUNCTION name (p_1: T_1; ....; p_m: T_m): T

---

where *name* specifies the name of the function, p_1 through p_m are formal input parameters with corresponding types T_1 through T_n and T is the type of the result value returned by the function. When the function is invoked, the input parameters are initialized by assignment of corresponding argument values.

Depending on the type of the parameter, this parameter passing protocol is equivalent to *call-by-value* (basic data types and the *Set*, *Sequence* and *Tuple* types) or *call-by-reference* (the *Entity* type).

The body of a routine consists of a sequence (possibly empty) local variable declarations followed by a sequence of IDL statements, and has the form:

```
VAR variable-declaration-sequence
BEGIN
 statement-sequence
END
```

Exceptions and error conditions can be handled at the user's level by explicitly testing for appropriate pre-conditions using the standard procedure/function parameter mechanism:

```
PROCEDURE P (IN x: X; OUT e-code: Integer; e-message: String)
  /* declare the procedure with the appropriate exception code and message
     parameters */

VAR a: X; code: INTEGER; mess: STRING;
BEGIN
 ...
P(a, code, mess);          /* procedure call includes the exception
                              code and message arguments */
IF ~(code = 0)             /* test exception code value and process condition */
  THEN PRINT mess;
 ...
END
```

Alternatively, a "SIGNALS *exception-name*" construct along with "exception handlers" (routines that trap exception signals and process according to user defined procedures) as in CLU can be used.

5.9.7. Type Checking

The declaration of a variable specifies its type; that is, the set of values it may denote. In an assignment, the value resulting from the evaluation of the expression must be within the domain of the type of the variable. There are no implicit type conversions in IDL. Thus IDL is a "type safe" language. It is not possible to treat a value of type *T* as if it were of another type. IDL is also statically typed in that type checking can be done

at compile time: that is, the type of the value that could result from the evaluation of any expression is known at compile-time. Hence every assignment can be type-checked to ensure that the variable can be assigned a value of the specified type. The type correctness rule for assignment is: (1) if the variable is an atomic type (a basic data type or abstract type *Entity* (of class *C*), then the expression being assigned must be the same basic data type or entity type (of any subclass of *C* as defined by the *IsA* hierarchy of entity classes); or (2) if the variable has type *Tuple*, *Set* or *Sequence*, then the expression must evaluate to a value of the corresponding type.

5.9.8. Statements

IDL is a statement-oriented language: statements are executed for their side effects and do not return any values. Assignments and procedure invocation are simple statements. *Control statements* direct the flow of control in an IDL program. The following control statements are provided:

If-then-else

This statement provides control flow selection and has the form:

IF *condition* THEN *statement_sequence-1* ELSE *statement_sequence-2* ENDIF

Here, *condition* is an expression of type *Boolean* and if it evaluates to TRUE, then the statement sequence following THEN is executed; otherwise the statement sequence following ELSE is executed. By nesting the *if-then-else* statements, a multiway control flow branching can be achieved.

While-do

This statement provides conditional iteration and has the form:

WHILE *condition* DO *statement_sequence* ENDWHILE

This repeatedly executes the *statement_sequence* as long as *condition* remains TRUE; the statement terminates when *condition* evaluates to FALSE.

For-do

This statement provides unconditional iteration and has the form:

FOR *loop_var* = *iterator* DO *statement_sequence* ENDFOR

An *iterator* iterates over the elements in its argument by returning individual elements on successive calls. It is implemented as a coroutine that saves its state between invocations. IDL has the following iterators:

(1) *Elems* - for *Set* and *Sequence* types

(2) *Chars* - for *String* type

(3) *From-to* and *From-to-by* - for *Integer* type

When the iterator is used in a For-do loop, in each iteration the loop variable is bound to the element returned by successive calls to the iterator as shown in the following example:

```
VAR s: {Gate};          /* s is a set of gates */
  q: Gate;          /* q is a (pointer to a) gate */
FOR q = Elems(s) DO   /* q is bound to successive elements in the set s */
  .....               /* do the desired gate operation */
ENDFOR
```

When the iterator signals the end of the sequence, the *for-do* statement terminates.

Break

This statement terminates the execution of the smallest enclosing *for-do* or *while-do* loop:

```
BREAK;
```

Import-database

This statement initializes the database program environment by reading in (from the specified directory) the database (schema + values) saved from a previous session:

```
IMPORT-DATABASE cpu FROM /vlsi/project/cpu
        /* retrieves database named "cpu" from directory "/vlsi/project/cpu" */
```

Define-database

This statement sets up a database program environment with the specified name; this is saved in the specified directory when the session is exited:

```
DEFINE-DATABASE cpu IN /vlsi/project/cpu
        /* defines database named "cpu" and saves in directory
        "/vlsi/project/cpu" upon exit from session */
```

Close-database

This statement terminates an interactive database session by saving the database environment in the directory (specified at the start of the session) and exiting:

```
CLOSE-DATABASE cpu   /* saves the database environment "cpu" in the specified
                        directory and exits */
```

The language also provides as statements the basic data model operations that modify the database.

5.9.9. Modularization

An IDL program consists of a group of database, procedure, function declarations and a sequence of IDL statements. *Database declarations* provide definitions of entity classes, variables and their types, and specifications of database operations. An environment is defined in terms of scoping units. IDL provides two levels of scoping. A *global scope* extends from a *define-database* or an *import-database* statement to a *close-database* statement. A *local scope* extends from the procedure/function header to the corresponding *end* (procedure/function) construct. Every local scope is nested within the global scope. There are as many local scopes as there are procedures and functions in the IDL program. Local scopes cannot be nested within each other. Variable declarations in the global scope are visible in the local scope, except when local declarations override global declarations with the same name.

5.10. Summary

This chapter has discussed the design issues of relevance to integrated database languages including data abstraction through composite and abstract data types, polymorphism and property inheritance, data persistence, type system implementation and system architecture. Several contemporary approaches to these issues are presented and compared in the context of different languages and systems proposed in recent literature. These ideas are then synthesized into a conceptual basis for the design of an integrated database language for a CAD database system based on the semantic data model presented in Chapter 4. The design of such a language called IDL is then described. The proposed language is an object-oriented procedural language that provides, in addition to the standard features of a general-purpose language, structured parameterizable types of *Tuple*, *Set* and *Sequence*, the abstract type *Entity* that models a design database object, and the data model concepts of entity classes, entity interrelationships, structural

abstraction hierarchies and bsic data model operations. Formal specifications of the semantic data model and IDL, and a prototype implementation of a structured value module are given in the next chapter.

Chapter 6

DESIGN DATABASE SPECIFICATION: AN ILLUSTRATIVE EXAMPLE

6.1. Introduction

In this chapter we shall present an illustrative case study involving the design and specification of a 4-bit adder using the design database system described in chapters 4 and 5. There are several different VLSI circuit design methodologies and techniques that can be used. A complete design specification involving all the different representations of the adder circuit, the design integrity constraints, and the analysis and synthesis tools used in various phases of the design would, of course, be very complex. For a complete and detailed information on VLSI design systems and techniques, the reader is referred to the following sources: [MeC80, Rub87, Seq83]. In this chapter, we shall focus on (1) the schematic (logic gate) and layout (transistor network) representations of a design; (2) the operations for creating a database instance containing the design and specification of a 4-bit adder; and (3) logic simulation and timing analysis as representative design processing tasks, and how they can be implemented using the design database system. The design methodology and some of the terminology is based on the ELECTRIC [Rub87] design system.

6.2. Database Scheme

The database scheme consists of a set of type, class, relationship and operation specifications. A *layout representation* of a VLSI circuit is defined in terms of *transistors, ports, contacts, poly wires* which constitute the layout primitives (blocks), and *cells* which are aggregations of components which may be layout primitives or other instances of predefined cells. Instances of the primitives are defined in terms of corresponding *instance entities* and many-to-1 *instance* relationships between instance and master enti-

ties. Transistors are the basic units of the VLSI circuit layout, and are modeled by the class *Trans*:

```
CLASS Trans = (          /* transistor layout primitive */
 ATTRIBUTE
  type: TransType,  /* transistor type; "TransType" is an enumerated type */
  ports: {PortInst}, /* transistor's ports (signal terminals) */
  struct: {Box},     /* mask geometry that defines the transistor */
  delay: INTEGER;/* propagation delay */
 CONSTRAINT
  k: KEY (type);
)
```

Transistor instances are specific placements of a transistor, and are modeled by the class *TransInst* and the instance relationship *I1*:

```
CLASS TransInst = (    /* transistor instance */
 ATTRIBUTE
  instID: INTEGER,     /* instance id */
  position: Point,     /* location of the transistor instance */
 CONSTRAINT
  k: KEY (instID);
)

INSTANCE I1 (TransInst[M], Trans[1])   /* M(any)-to-1 instance relationship
                                          between transistor instance and
                                          transistor entities */
```

Ports define connection points on a transistor, and are modeled by the class *port*:

```
CLASS Port = (                 /* port layout primitive */
 ATTRIBUTE
  name: STRING,                /* port name */
  struct: Box,                 /* mask geometry */
  direction: BOOLEAN;          /* 0=in, 1=out */
 CONSTRAINT
  k: KEY (name);
)
```

Port instances are specific placements of a port, and are modeled by the class *PortInst* and the instance relationship *I2*:

```
CLASS PortInst = (      /* port instance */
 ATTRIBUTE
  instID: INTEGER,      /* instance id */
  position: Point,      /* location of the port instance */
  signal: Signal;       /* logic signal carried by the port */
 CONSTRAINT
  k: KEY (instID);
)

INSTANCE I2 (PortInst[M], Port[1])      /* instance relationship between
                                           port instance and port entities */
```

Contacts are circuit layout elements that provide connection points into the VLSI circuit chip from the external world, and are modeled by the class *Contact*:

```
CLASS Contact = (       /* contact layout primitive */
 ATTRIBUTE
  type: ContactType,    /* contact type; "ContactType" is an
                           enumerated type */
  ports: {PortInst},    /* transistor ports connected */
  struct: {Box};        /* mask geometry */
 CONSTRAINT
  k: KEY (type);
)
```

Contact instances are specific placements of a contact, and are modeled by the class *ContactInst* and the instance relationship *I3*:

```
CLASS ContactInst = (  /* contact instance */
 ATTRIBUTE
  instID: INTEGER,      /* instance id */
  position: Point,      /* location of the contact */
 CONSTRAINT
  k: KEY (instID);
)

INSTANCE I3 (ContactInst[M], Contact[1])      /* instance relationship between
                                                 contact instance and contact
                                                 entities */
```

Poly wires define connections that carry signals between port instances, and are modeled by the class *PolyWire*:

```
CLASS PolyWire = (    /* poly wire layout primitive */
 ATTRIBUTE
  from: PortInst,     /* from port instance */
  to: PortInst,       /* to port instance */
  delay: INTEGER;     /* propagation delay */
 CONSTRAINT
  k: KEY (from, to);
)
```

The *mask geometry* of the VLSI circuit is defined in terms of the graphics primitives *box* (modeled by the class *Box*) and *point* (modeled by the class *Point*):

```
CLASS Box = (         /* box graphics primitive (mask geometry) */
 ATTRIBUTE
  width: INTEGER,     /* box width (x-axis) */
  height: INTEGER,    /* box height (y-axi) */
  llCorner: Point,    /* box's lower left corner */
  layer: LayerType;   /* box's mask layer; "LayerType" is an enumerated type */
 CONSTRAINT
  k: KEY (llCorner, layer);
)

CLASS Point = (       /* point primitive */
 ATTRIBUTE
  xCoord: INTEGER,    /* x-coordinate */
  yCoord: INTEGER;    /* y-coordinate */
 CONSTRAINT
  k: KEY (xCoord, yCoord);
)
```

Cells are user-defined aggregations of instances of the layout primitives, and are modeled by the class *Cell*:

```
CLASS Cell = (              /* cell layout primitive */
 ATTRIBUTE
  name: STRING,             /* cell name */
  struct: {Element};        /* cell composition structure */
  exPorts: <PortInst>;      /* exported (interface) ports of the cell */
 CONSTRAINT
  k: KEY (name);
)
```

The structure of a cell is defined by a set of elements (cell, transistor, port, contact instances, and poly wires) which are modeled by the class *Element*:

```
CLASS Element = (              /* element layout primitive */
 CONSTRAINT
  m: UNION (CellInst, TransInst, PortInst, ContactInst, PolyWire);
)
```

Cell instances are specific placements (defined by a transformation matrix) of a cell, and are modeled by the class *CellInst* and the instance relationship *I4*:

```
CLASS CellInst = (             /* cell instance */
 ATTRIBUTE
  instID: INTEGER,             /* instance id */
  transMatrix: <INTEGER>;      /* cell placement transformation matrix */
 CONSTRAINT
  k: KEY (instID);
)

INSTANCE I4 (CellInst[M], Cell[1])      /* instance relationship between cell
                                           instance and cell entities */
```

A *library* is a user-defined collection of cells:

```
CLASS Library = (              /* cell library */
 ATTRIBUTE
  name: STRING,                /* library name */
  collection: {Cell};          /* library cell collection */
 CONSTRAINT
  k: KEY (name);
)
```

The semantic integrity constraints on a layout include *design rules* (spacings, clearances, overlaps and orientations etc.) and *connectivity* (all poly wires interconnect appropriate port instances, and no dangling wires).

A *schematic representation* is defined in terms of *logic gates* (AND, OR, XOR etc.), *function blocks* (adders, registers etc.), and *logic signals* which constitute the schematic primitives.

```
CLASS Signal = (               /* logic signal schematic primitive */
 ATTRIBUTE
  name: STRING,                /* signal name */
  value: BOOLEAN;              /* logic value */
 CONSTRAINT
  k: KEY (name);
)
```

```
CLASS AND = (          /* AND gate schematic primitive */
 ATTRIBUTE
  in: {Signal},        /* input signals */
  out: Signal,         /* output signal */
  delay: INTEGER;      /* gate delay */
 CONSTRAINT
  k: KEY (in, out);
)

CLASS OR = (           /* OR gate schematic primitive */
 ATTRIBUTE
  in: {Signal},        /* input signals */
  out: Signal,         /* output signal */
  delay: INTEGER;      /* gate delay */
 CONSTRAINT
  k: KEY (in, out);
)

CLASS XOR = (          /* exclusive OR gate schematic primitive */
 ATTRIBUTE
  in: {Signal},        /* input signals */
  out: Signal,         /* output signal */
  delay: INTEGER;      /* gate delay */
 CONSTRAINT
  k: KEY (in, out);
)
```

The database is further structured in terms of design technologies and environments. A *technology* is a collection of layout, schematic or other primitives.

```
CLASS Technology = (       /* VLSI design technology */
 ATTRIBUTE
  techGroup: STRING,       /* technology group name */
  groupSpec: TechGroup;    /* technology group */
 CONSTRAINT
  k: KEY (techGroup);
)

CLASS TechGroup = (        /* technology group */
 CONSTRAINT
  m: UNION (LayoutGroup, GraphicsGroup, SchematicGroup);
)
```

A *layout group* is a collection of layout primitives:

```
CLASS LayoutGroup = (              /* layout group */
 ATTRIBUTE
  name: STRING,                    /* layout group name */
  collection: {LayoutBlock};       /* layout primitives collection */
 CONSTRAINT
  k: KEY (name);
)

CLASS LayoutBlock = (              /* layout primitive */
 CONSTRAINT
  m: UNION (Trans, Port, Contact, PolyWire);
)
```

A *graphics group* is a collection of graphics primitives:

```
CLASS GraphicsGroup = (            /* graphics group */
 ATTRIBUTE
  name: STRING,                    /* graphics group name */
  collection: {GraphicsBlock};     /* graphics primitives collection */
 CONSTRAINT
  k: KEY (name);
)

CLASS GraphicsBlock = (            /* graphics primitive */
 CONSTRAINT
  m: UNION (Box, Point);
)
```

A *schematic group* is a collection of schematic primitives:

```
CLASS SchematicGroup = (           /* schematic group */
 ATTRIBUTE
  name: STRING,                    /* schematic group name */
  collection: {SchematicBlock};    /* schematic primitives collection */
 CONSTRAINT
  k: KEY (name);
)

CLASS SchematicBlock = (                      /* schematic primitive */
 CONSTRAINT
  m: UNION (AND, OR, XOR, NAND, INV);         /* AND, OR etc. gates */
)
```

An *environment* is a collection of technologies:

```
CLASS Environment = (        /* vlsi design environment */
 ATTRIBUTE
  name: STRING,              /* environment name */
  collection: {Technology};  /* collection of technologies */
 CONSTRAINT
  k: KEY (name);
)
```

The *Has-comp* and *Is-a* class interrelationship hierarchies defined by the definitions above are shown in Figure 6.1.

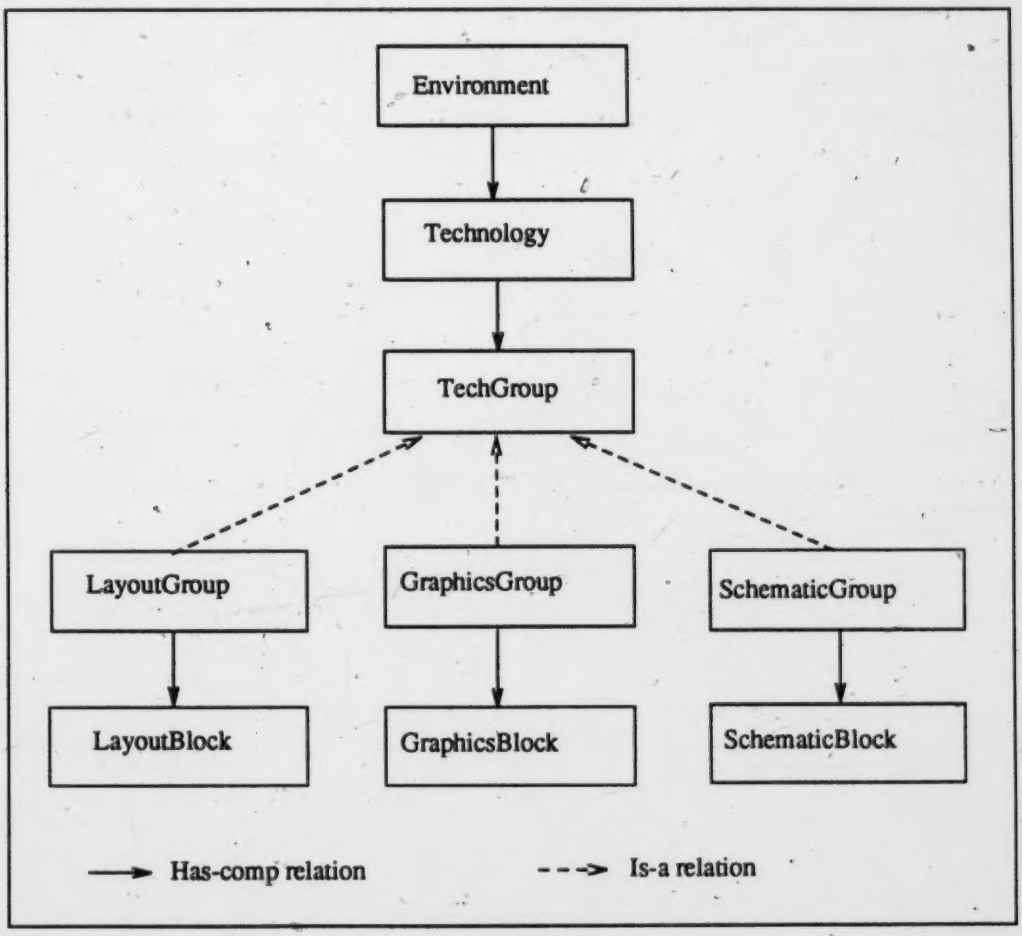

Figure 6.1(a)

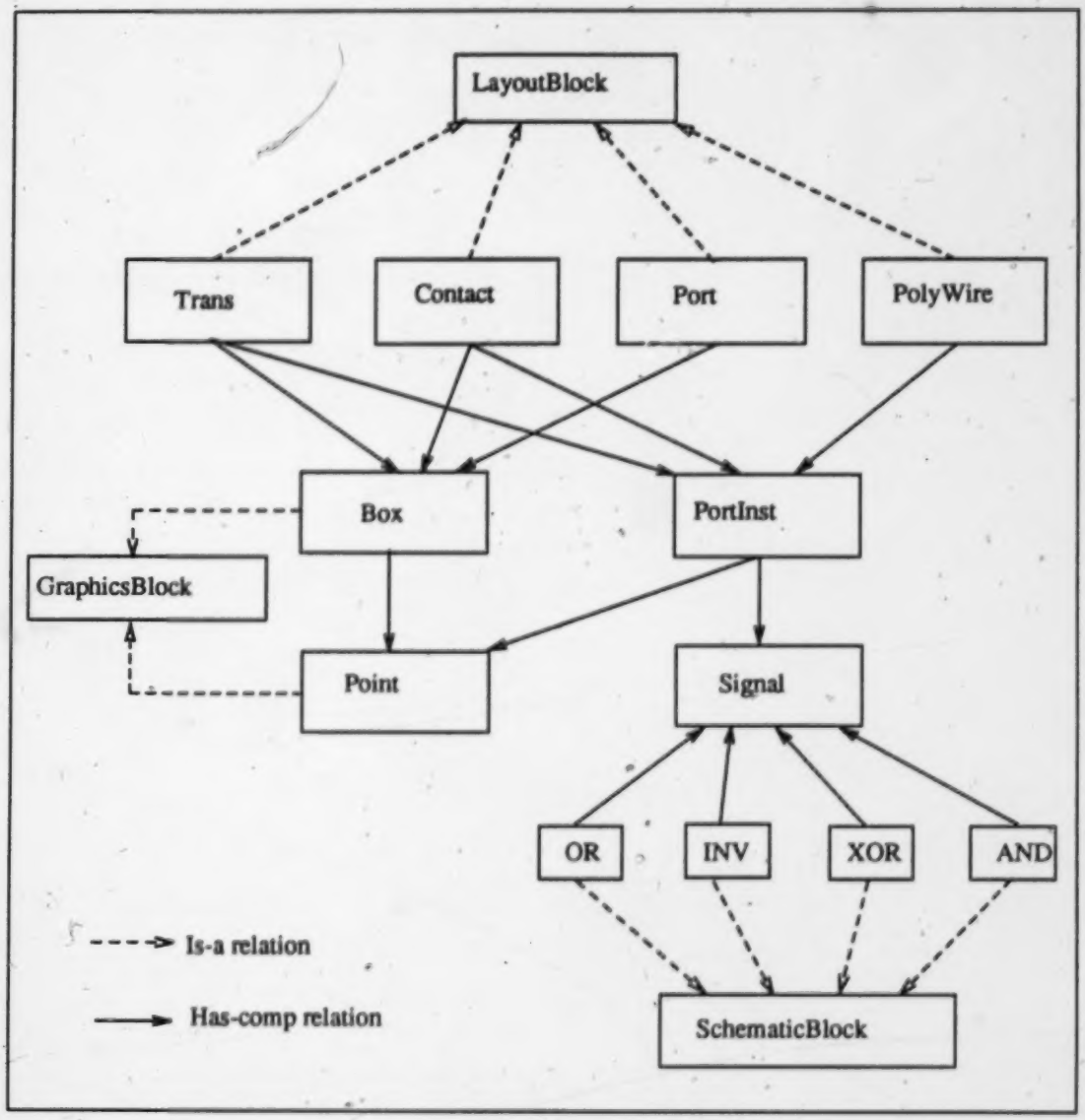
LayoutBlock
Trans
Contact
Port
PolyWire
Box
PortInst
GraphicsBlock
Point
Signal
OR
INV
XOR
AND
Is-a relation
Has-comp relation
SchematicBlock

Figure 6.1(b)

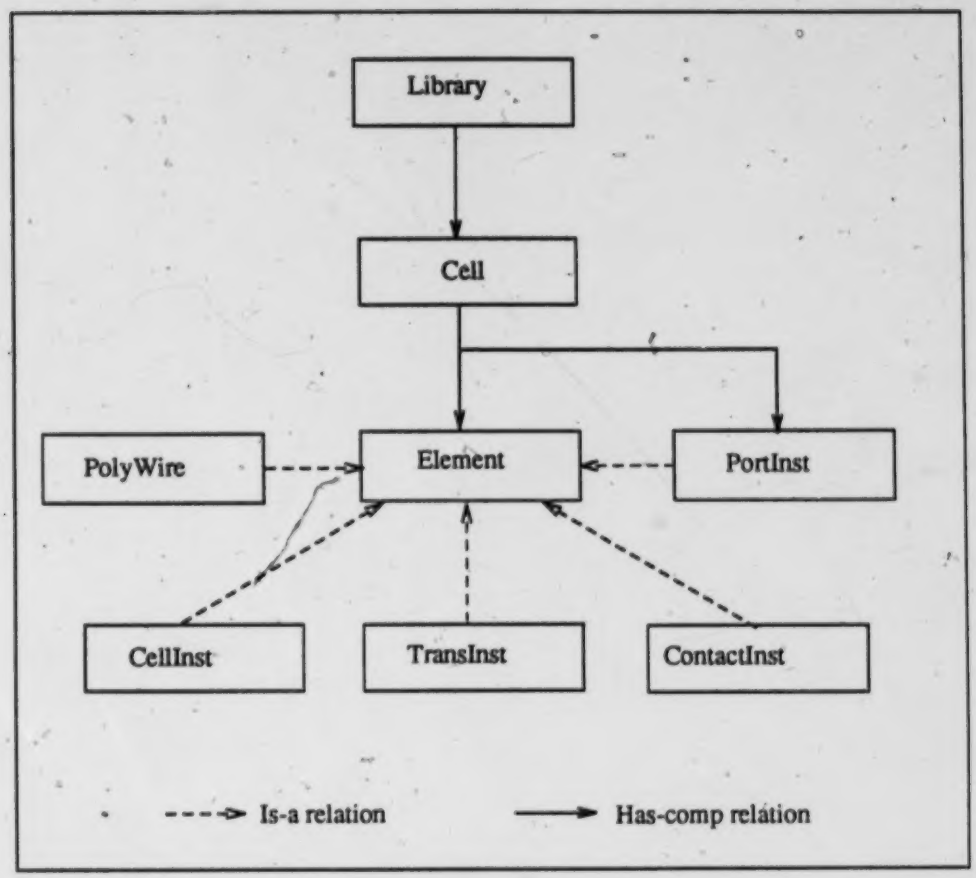

Figure 6.1(c)

Figure 6.1 *Has-comp* and *Is-a* Class Interrelationship Hierarchies

6.3. An Utility Library

An *utility library* of procedures and functions provides several useful operations that can be used in conjunction with the class definitions given in the previous section. We describe some of these operations that are of immediate relevance; other more specialized ones can be defined by the user.

• *MakeBox* creates a box with specified attribute values:

```
FUNCTION MakeBox (width, height: INTEGER; llCorner: Point; layer: LayerType): Box

VAR b: Box;

BEGIN
  b = CreateEnt(Box);
  SET width(b) = width;
  SET height(b) = height;
  SET llCorner(b) = llCorner;
  SET layer(b) = layer;
  RETURN (b);
END
```

• *MakePoint* creates a point with specified x and y coordinates:

```
FUNCTION MakePoint (xCoord, yCoord: INTEGER): Point

VAR p: Point;

BEGIN
  p = CreateEnt(Point);
  SET xCoord(p) = xCoord;
  SET yCoord(p) = yCoord;
  RETURN (p);
END
```

• *MakeTrans* creates a transistor from given interface and structure specifications:

```
FUNCTION MakeTrans (type: TransType; ports: {PortInst}; struct: {Box}): Trans

VAR t: Trans;

BEGIN
  t = CreateEnt(Trans);
  SET type(t) = type;
  SET ports(t) = ports;
  SET struct(t) = struct;
  RETURN (t);
END
```

• *MakePort* creates a port from given interface and structure specifications:

```
FUNCTION MakePort (name: STRING; struct: Box, direction: BOOLEAN): Port

VAR p: Port;

BEGIN
  p = CreateEnt(Port);
  SET name(p) = name;
  SET struct(p) = struct;
  SET direction(p) = direction;
  RETURN (p);
END
```

• *MakeContact* creates a contact from given interface and structure specifications:

```
FUNCTION MakeContact (type: ContactType; ports: {PortInst}; struct: {Box}): Contact
VAR c: Contact;

BEGIN
  c = CreateEnt(Contact);
  SET type(t) = type;
  SET ports(t) = ports;
  SET struct(t) = struct;
  RETURN (c);
END
```

• *MakePortInst* creates a port instance of a given port at a specified position:

```
FUNCTION MakePortInst (p: Port; position: Point): PortInst

VAR i: PortInstance;

BEGIN
  i = CreateEnt(PortInst);
  SET instID(i) = GetUniqueID();
  SET position(i) = position;
  CreateInst(I2, p, i);
  RETURN (i);
END
```

• *MakePolyWire* creates a poly wire connecting specified port instances:

```
FUNCTION MakePolyWire (from, to: PortInst): PolyWire

VAR w: PolyWire;

BEGIN
  w = CreateEnt(PolyWire);
  SET from(w) = from;
  SET to(w) = to;
  RETURN (w);
END
```

• *MakeCell* creates a cell with specified name and structure:

```
FUNCTION MakeCell (name: STRING; struct: {Element}; exPorts: <PortInst>): Cell

VAR c: Cell;

BEGIN
  c = CreateEnt(Cell);
  SET name(c) = name;
  SET struct(c) = struct;
  SET exPorts(c) = exPorts;
  RETURN (c);
END
```

• *MakeSignal* creates a signal with the specified name:

```
FUNCTION MakeSignal (name: STRING): Signal

VAR s: Signal;

BEGIN
  s = CreateEnt(Signal);
  SET name(s) = name;
  RETURN (s);
END
```

• *MakeXOR* creates a XOR gate with specified name and interface signals:

```
FUNCTION MakeXOR (name: STRING; in1, in2, out: Signal): XOR

VAR x: XOR;

BEGIN
  x = CreateEnt(XOR);
  SET name(x) = name;
  SET in(x) = {in1, in2};
  SET out(x) = out;
  RETURN (x);
END
```

Other operations that create entities in other classes and manipulate them can be similarly defined.

6.4. Defining a 4-bit Adder

The schematic representation of a 4-bit adder is defined in terms of the class *FourBitAdder*:

```
CLASS FourBitAdder = (      /* 4-bit adder */
 ATTRIBUTE
  name: STRING,             /* adder name */
  x: <Signal>,              /* input 1 (4 bits) */
  y: <Signal>,              /* input 2 (4 bits) */
  sum: <Signal>,            /* sum (4 bits) */
  cIn: Signal,              /* carry in */
  cOut: Signal,             /* carry out */
  struct: <FullAdder>;      /* composition structure */
 CONSTRAINT
  k: KEY (name);
  r: RANGE (Length(x)=4 & Length(y)=4 & Length(sum)=4 & Length(struct)=4);
)
```

The *interface* of the 4-bit adder is defined by the signals *x*, *y*, *sum*, *cIn* and *cOut*, while its *structure* is defined by an array (of size 4) of *full adders* (Figure 6.2).

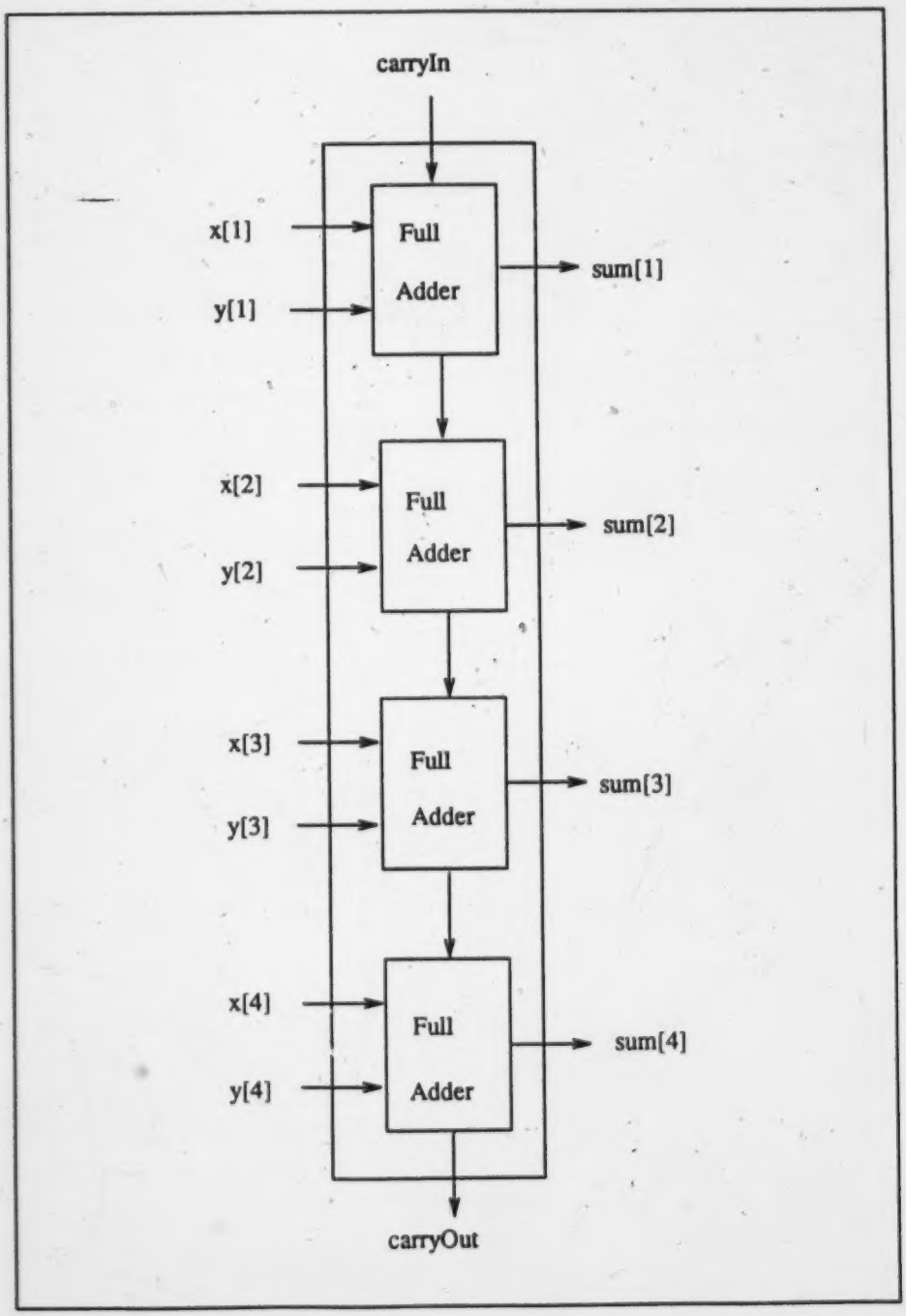

Figure 6.2 4-bit Adder: VLSI Circuit Schematic

The schematic representation of the full adder is defined in terms of the class *FullAdder*:

```
CLASS FullAdder = (        /* full adder */
 ATTRIBUTE
  name: STRING,            /* adder name */
  x: Signal,                    /* input 1 */
  y: Signal,                    /* input 2 */
  sum: Signal,             /* sum */
  cIn: Signal,             /* carry in */
  cOut: Signal,            /* carry out */
  struct: {SchematicBlock};      /* composition structure */
 CONSTRAINT
  k: KEY (name);
)
```

The interface of the full adder is defined by the signals *x*, *y*, *sum*, *cIn* and *cOut*, while its structure is defined by a set of schematic blocks (half adders and XOR gate) (Figure 6.3).

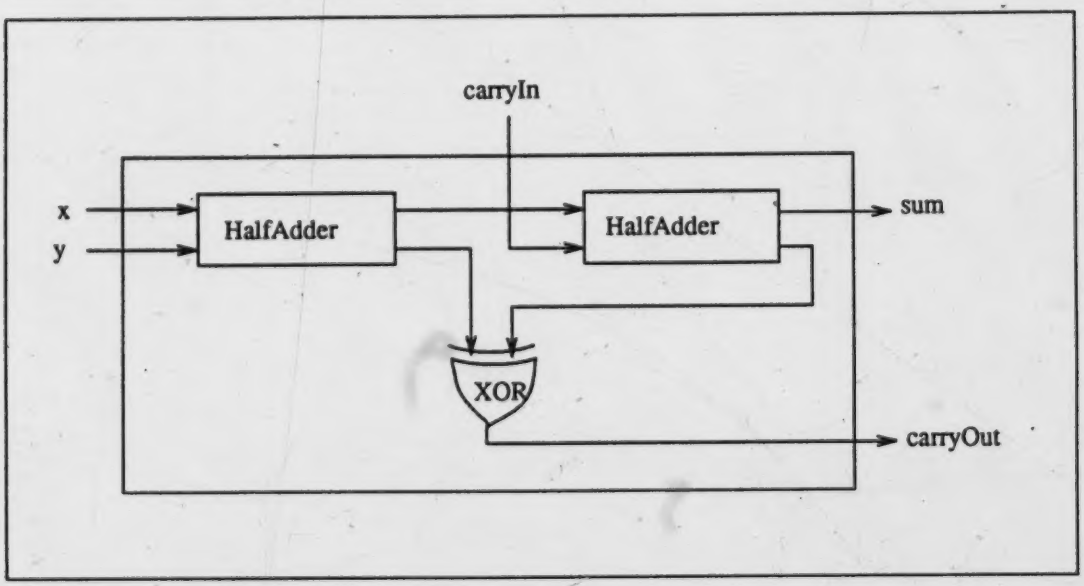

Figure 6.3 Full Adder: VLSI Circuit Schematic

The schematic representation of the half adder is defined in terms of the class *HalfAdder*:

```
CLASS HalfAdder = (              /* half adder */
 ATTRIBUTE
  name: STRING,                  /* name */
  x: Signal,                     /* input 1 */
  y: Signal,                     /* input 2 */
  sum: Signal,                   /* sum */
  carry: Signal,                 /* carry */
  struct: {SchematicBlock};      /* composition structure */
 CONSTRAINT
  k: KEY (name);
)
```

The interface of the half adder is defined by the signals *x*, *y*, *sum*, and *carry*, while its structure is defined by a set of schematic blocks (NAND, INV and XOR gates) (Figure 6.4).

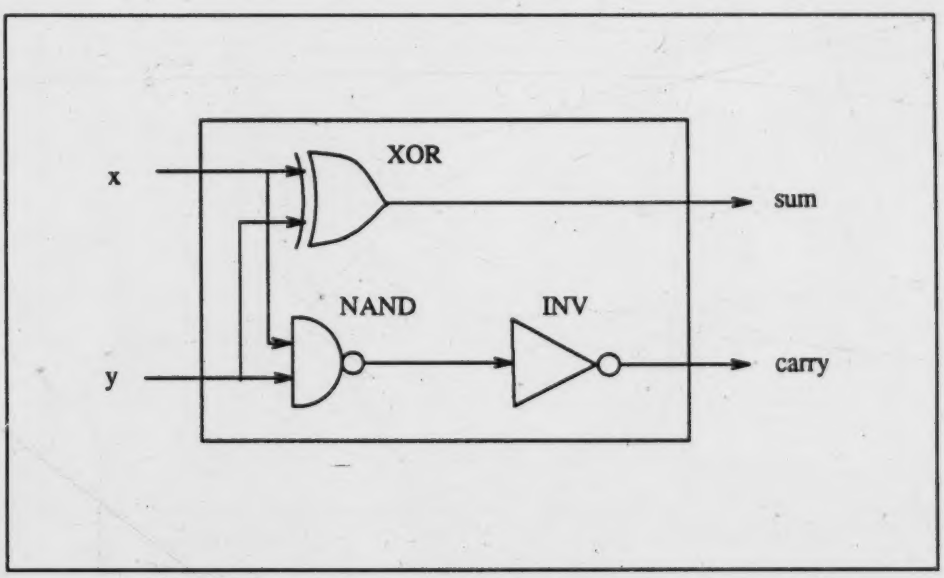

Figure 6.4 Half Adder: VLSI Circuit Schematic

The *Has-comp* class interrelationship hierarchy defined by the above specifications is shown in Figure 6.5.

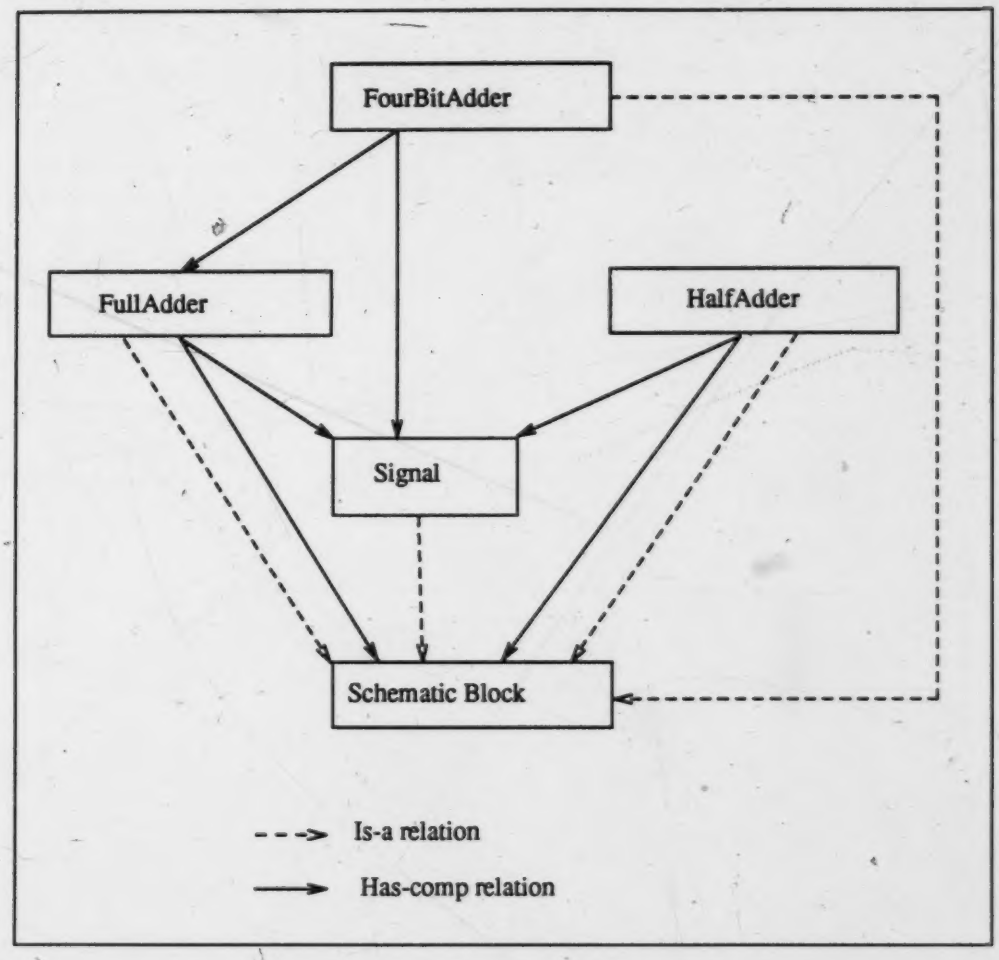

Figure 6.5 4-bit Adder: *Has-comp* and *Is-a* Class Interrelationship Hierarchy

To create a database instance of the schematic representation of a 4-bit adder we define the following procedures and functions (in IDL).

• *MakeFourBitAdder* creates a 4-bit adder entity from given interface specifications:

```
FUNCTION MakeFourBitAdder (name: STRING; x, y, sum: <Signal>;
                           cIN, cOut: Signal): FourBitAdder

VAR f: FourBitAdder; l: <FullAdder>; c: <Signal>;
  uName: <STRING>; u: FullAdder;

BEGIN
  f = CreateEnt(FourBitAdder);
  SET name(f) = name;
  SET x(f) = < x[1], x[2], x[3], x[4]>;
  SET y(f) = < y[1], y[2], y[3], y[4]>;
  SET sum(f) = < sum[1], sum[2], sum[3], sum[4]>;
  SET cIn(f) = cIn;
  SET cOut(f) = cOut;
  l = emptySeq();
  uName = InputFullAdderName(name, 1);
  u = MakeFullAdder(uName, x[1], y[1], sum[1], cIn, c[1]);
  Append(l, u);
  uName = InputFullAdderName(name, 2);
  u = MakeFullAdder(uName, x[2], y[2], sum[2], c[1], c[2]);
  Append(l, u);
  uName = InputFullAdderName(name, 3);
  u = MakeFullAdder(uName, x[3], y[3], sum[3], c[2], c[3]);
  Append(l, u);
  uName = InputFullAdderName(name, 4);
  u = MakeFullAdder(uName, x[4], y[4], sum[4], c[3], cOut);
  Append(l, u);
  SET struct(f) = l;
  RETURN (f);
END
```

• *MakeFullAdder* creates a full adder entity from given interface specifications:

```
FUNCTION MakeFullAdder (name: STRING; x, y, sum, cIN, cOut: Signal): FullAdder

VAR f: FourBitAdder; s: <Signal>;
  a1, a2: HalfAdder; xor: XOR; aName, xName: STRING;

BEGIN
  f = CreateEnt(FullAdder);
  SET name(f) = name;
  SET x(f) = x;
  SET y(f) = y;
  SET sum(f) = sum;
  SET cIn(f) = cIn;
  SET cOut(f) = cOut;
  aName = InputHalfAdderName(name, 1);
  a1 = MakeHalfAdder(aName, x, y, s[1], s[2]);
  aName = InputHalfAdderName(name, 2);
  a2 = MakeHalfAdder(aName, s[1], cIn, sum, s[3]);
  xName = InputXORName(name);
  xor = MakeXOR(xName, s[2], s[3], cOut);
  SET struct(f) = { a1, a2, xor };
  RETURN (f);
END
```

• *MakeHalfAdder* creates a half adder entity from given interface specifications:

```
FUNCTION MakeHalfAdder (name: STRING; x, y, sum, carry: Signal): HalfAdder

VAR h: HalfAdder; s: Signal; xor: XOR; n: NAND; i: INV;
  gName: STRING;

BEGIN
  h = CreateEnt(HalfAdder);
  SET name(h) = name;
  SET x(h) = x;
  SET y(h) = y;
  SET sum(h) = sum;
  SET carry(h) = carry;
  gName = InputGateName(name, "XOR");
  xor = MakeXOR(gName, x, y, sum);
  gName = InputGateName(name, "NAND");
  n = MakeNAND(gName, x, y, s);
  gName = InputGateName(name, "INV");
  i = MakeINV(gName, s, carry);
  SET struct(f) = { xor, n, i };
  RETURN (h);
END
```

To create a layout representation of the adder defined above, first we use a sequence of make functions that create the appropriate transistors, ports, poly wires etc. Then a cell that implements the 4-bit adder composed from these layout primitives is defined. The "exported" ports of this cell are those that carry the interface signals used in defining the schematic representation of the 4-bit adder. A correspondence between the cell (layout) and the functional four bit adder (schematic) is now set up in terms of a representation relationship *R1* defined as follows:

REPRESENTATION R1 (Cell[M], SchematicBlock[1])

An instance of this representation relationship is created and and the interface signals of the 4-bit adder are assigned to the exported ports of the cell by means of a *SetUpRep* procedure as follows:

```
PROCEDURE SetUpRep (IN relName: REPRESENTATION;
                    c: Cell, s: SchematicBlock; interface: <Signal>)

VAR i: INTEGER; p: PortInst;

BEGIN
  CreateRep(relName, c, s);
  FOR i = 1 TO Length(exPorts(c)) DO
    p = (exPorts(c))[i];
    SET signal(p) = interface[i];
  ENDFOR
END
```

In the procedure above, the parameters *relName* is the representation relationship name *Ri*; *c* is the cell that defines the layout representation of the 4-bit adder *s*; *exPorts* is the sequence of exported port instances in the cell that correspond to the sequence of interface signals *interface* defining the 4-bit adder.

6.5. Design Database Processing

VLSI design involves processing design data by several different *design tools* at various stages in the design process. A number of these tools have been developed for static and dynamic analysis, simulation, verification and optimization of the VLSI circuit being designed. Each of these tools expects its input and produces its output in specified structured formats. Hence, before a design tool can be invoked to process the design data, it is necessary to create the appropriate input data structures for it. These structures can be considered as defining external-level *views* on the stored data in the design database. Similarly, the output generated by the design tool is another external view which has to be converted back into the internal structure to update the database. Examples of design tools include design rule checkers, electrical rule checkers, test generators, routers, automatic layout generators, placement tools, silicon compilers, circuit-level simulators (SPICE [Nag75], switch-level simulators (RSIM [Ter83], RNL [BaT80] etc.), logic

simulators (CADAT [HHB85], MARS [Sin83]) etc.), timing analyzers, and functional verifiers etc. In the following sections we discuss timing analysis and gate-level logic simulation techniques, and illustrate how these tasks can be implemented using our design database system.

6.5.1. Logic Simulation

Logic simulation consists of determining circuit behaviour as a function of time. This technique is appropriate when it is impossible or prohibitively expensive to derive a closed-form analytical solution describing circuit behaviour. Essentially, logic simulation involves computing the circuit output (signal values at specified output terminals) as a function of particular circuit input called a *test vector* (signal values at specified input terminals). Logic simulation can be performed at the *switch level* (circuit elements and signals are modeled, respectively, by transistors and analog waveforms) or at the *gate level* (circuit elements and signals are modeled, respectively, by gates and Boolean values). Gate-level simulation is computationally much less expensive than switch-level simulation. The circuit model for gate-level logic simulation consists of elements (gates with specified signal propagation delays), interconnections (zero-delay wires), and logic signal values 0 (false), 1 (true) and x (undefined). The behaviour of each gate is defined by a truth table extended to include the third logic value x. The circuit being simulated is defined in terms of its schematic representation consisting of instances of these gates, each with specified input and output signals. The objective of the simulation is to compute the signal values at the outputs of selected gates in response to a specified test vector applied at the inputs of selected gates.

The actual simulation itself can be either *incremental-time simulation* or *event-driven simulation*. In incremental simulation, at each simulation step, the simulation clock is advanced by a fixed increment determined by the smallest propagation delay of

the individual model elements, and all the signal values are recalculated. Event-driven simulation consists of a sequence of "events", each of which is a change in value at a particular time of a specified signal. At each simulation step, the system removes the "next event" from a time-ordered priority queue of events, calculates signal value changes that will occur as a result of these events at specified future times, and schedules these changes by entering them into the event queue. The simulation is then advanced another step. This process is repeated until a specified time or until the event queue is empty when the simulation completes and the desired output signal values can be read off.

To apply the gate-level logic simulation to the 4-bit adder defined in our database, the schematic representation can be extracted from the database and restructured to conform to the input format requirements of a simulator like RSIM [Ter83], and the simulator invoked to process this data. Alternatively, an implementation of a gate-level logic simulation can be defined in our system by the specifications shown in Figure 6.6 below:

```
TYPE Event :: signal: Signal;  /* signal entity */
        value: BOOLEAN;        /* logic value of the signal */
        time: INTEGER;         /* time at which the signal value changes */

VAR  eq: <Event>;              /* event queue - time-ordered priority queue
                                  of events implemented as a heap array */
   currentTime: INTEGER;       /* current time */
   simTime: INTEGER;           /* desired final simulation time */

/* main procedure */

PROCEDURE LogicSimulation()

BEGIN
  eq = InitEQ();                          /* initialize event queue */
  currentTime = InputCurrentTime();       /* read in current time */
  simTime = InputSimTime();               /* read in simulation time */
  Simulate (currentTime, simTime, eq);    /* event-driven simulation */
END

/* event queue initialization function */

FUNCTION InitEQ(): <Event>

VAR eq: <Event>;                          /* event queue */
   event: Event; flag: BOOLEAN; n: INTEGER;

BEGIN
  eq = emptySeq();                /* create an empty sequence of events */
  InputEvent (event, flag);       /* read in initial events; flag = TRUE => eof */
  WHILE (flag) DO
    Append(eq, event);            /* append event to event sequence */
    InputEvent (event, flag);     /* read in initial event */
  ENDWHILE
  n = Length(eq);                 /* length of initial event sequence */
  eq = MkPriorityEQ(eq, 1, n); /* create a time-ordered priority queue of events */
END
```

--->

```
/* event-driven simulation procedure */

PROCEDURE Simulate (IN currentTime, simTime: INTEGER; eq: <Event>)

VAR event: Event;

BEGIN
 /* while current time < simulation time and event queue is not empty */
 WHILE ((currentTime < simTime) & (~(IsEmpty(eq)))) DO
  event = GetNextEvent (currentTime);     /* get next event from the event queue */
  currentTime = time(event);              /* advance current time */
  CalcChanges (event);                    /* calculate signal value changes */
 ENDWHILE
END

/* calculating signal value changes (future events) for a specified current
      event */

PROCEDURE CalcChanges (IN event: Event)

VAR s: Signal; v: BOOLEAN; e: Event; g: Gate; gates: {Gate};

BEGIN
 s = signal(event);       /* signal defined by the event */
 v = value(event);        /* logic value defined by the event */

 /* retrieve all gates which have the specified signal as an input */
 gates = RETRIEVE ALL g: Gate WHERE (s IN in(g));

 /* iterate over the set of gates retrieved */
 FOR g = Elems(gates) DO
  e = ComputeOutput(g, event);      /* compute output signal value of the gate */
  IF ( ~(signal(e) = NULL) )        /* if the signal value has changed from its
                                    current value */
   THEN InsertEQ(e, eq);    /* schedule future event by inserting into the event queue */
  ENDIF
 ENDFOR
END
```

--->

```
/* computing a future event that changes the value of an output signal of a
      gate as a result of a current event */

FUNCTION ComputeOutput (g: Gate; event: Event): Event

VAR e: Event;

BEGIN
 IF (g IN Ext(AND))                    /* check if gate is an AND gate */
  THEN e = ComputeAND(g, event);       /* compute output signal change */
 ENDIF
 IF (g IN Ext(OR))                     /* check if gate is an OR gate */
  THEN e = ComputeOR(g, event);        /* compute output signal change */
 ENDIF
 IF (g IN Ext(INV))                    /* check if gate is an INV gate */
  THEN e = ComputeINV(g, event);       /* compute output signal change */
 ENDIF
 ...                                   /* similarly for other gates */
 RETURN(e);                            /* return future event */
END
```

Figure 6.6 Logic Simulation: Data Structures and Algorithms

Type *Event* defines the data structure for an event - a signal value change at a specified time. Variable *eq* is time-ordered priority queue of events which is implemented as a min-heap array; *currentTime* and *simTime* are, respectively, the current time and the desired final simulation time. The procedure *LogicSimulation* is the main procedure. Procedure *InitEQ* initializes the event queue by reading in the specified initial test vector. Procedure *MkPriorityEQ* creates a time-ordered priority queue of events. Procedure *Simulate* works by getting the next event from the event queue (*GetNextEvent*) and calculating the changes (*CalcChanges*) in signal values triggered by this event. Procedure *CalcChanges* retrieves all gates whose inputs contain the signal defined by the event, and computes the change in gate's output (*ComputeOutput*). Function *ComputeOutput* determines the type of gate and invokes a corresponding procedure to compute the output of the gate - a future event to be scheduled if it is different from the current output. The

Simulate procedure loops until the current time exceeds the simulation time or the event queue is empty. At this point, the desired output signal values can be read off.

6.5.2. Timing Analysis

Timing analysis consists of determining the longest-delay signal propagation path from the input to the output terminals in a circuit. This is useful in ensuring correctness of clock cycles and in optimizing some performance measures. For the purpose of timing analysis, the VLSI circuit is modeled as a network of *nodes* (ports or terminals at which signals appear) interconnected by *arcs* (circuit elements - transistors and wires - with specified signal propagation delays). Signals propagate from the input nodes to the output nodes along many different paths with corresponding delays. Timing analysis has to determine the overall circuit delay as the delay along the longest-delay path to some output node. The basic technique for determining the longest-delay path is defined by the following recursive formula:

---

circuit_delay = max { longest_delay ($NODE_i$)) }

longest_delay (NODE) =
    if NODE is an input node
        then 0
        else max { longest_delay(pred(NODE)) + delay(ARC)) }

---

where *i* ranges over the output nodes, *pred* is a function that returns the set of predecessor nodes of its argument, ARC is the sequence of circuit elements connecting a predecessor node to the node NODE, and *delay*(ARC) is the sum of delays of the elements in the sequence ARC. After the longest delay path is determined a more accurate timing verification can be done by applying circuit-level simulation to this path.

The longest-delay calculations for our 4-bit adder database can be implemented as

shown in the specifications in Figure 6.7 below:

```
TYPE ArcType = {"wire", "transistor"}  /* arc type enumerated type */

   /* network arc */
   Arc :: source: PortInst;              /* source port instance */
        dest: PortInst;                  /* destination port instance */
        arcType: ArcType;                /* arc type */
        delay: INTEGER;                  /* arc delay */

   /* network node */
   Node :: port: PortInst;               /* port instance */
         propDelay: INTEGER;             /* propagation delay from a source
                                         up to this node */

VAR  arcs: <Arc>;        /* network arcs table */
   nodes: <Node>;        /* topologically sorted network nodes list */

/* main procedure */

PROCEDURE  CalculateDelay()

BEGIN
  arcs = SetUpArcs();              /* set up the arcs table */
  nodes = TopoSort(arcs);          /* sort network nodes topologically */
  LongestDelay (arcs, nodes);      /* compute propagation delay to each node */
END
```

--->

```
/* function to set up the network arcs table */

FUNCTION SetUpArcs(): <Arc>

VAR arc: Arc; arcs: <Arc>; arcType: ArcType;
   p, s, d: PortInst; ports, sources, dests: {PortInst};
   m: Port; t: Trans; tI: TransInst;
   w: PolyWire; delay: INTEGER; flag: BOOLEAN;

BEGIN
 arcs = emptySeq();                          /* initialize arcs table */
 FOR w = Elems (Ext(PolyWire)) DO            /* iterate over the wires in the circuit */
  s = from(w);                               /* source port instance */
  d = to(w);                                 /* destination port instance */
  arcType = "wire";                          /* arc is a wire */
  delay = delay(w);                          /* arc delay */
  arc = mk-Arc(s, d, arcType, delay);        /* make an arc entity */
  Append(arcs, arc);                         /* enter into arcs table */
 ENDFOR
 FOR tI = Elems (Ext(TransInst)) DO          /* iterate over transistor instances */
  t = GetMaster(I1, tI);                     /* get master transistor */
  ports = ports(t);                          /* get its ports */
  sources = emptySet();                      /* initialize source port set */
  dests = emptySet();                        /* initialize destination port set */
  FOR p = Elems(ports) DO                    /* iterate over the ports */
   m = GetMaster(I2, p);
   IF (direction(m))                         /* FALSE (0) = source port;
                                              TRUE (1) = destination port */
    THEN Insert(dests, p);                   /* insert port instance as a source */
    ELSE Insert(sources, p);                 /* insert port instance as a destination */
   ENDIF
  ENDFOR
  arcType = "transistor";                    /* arc is a transistor */
  delay = delay(t);                          /* arc delay */
  FOR s = Elems(sources) DO                  /* iterate over source port
                                              instances */
   FOR d = Elems(dests) DO                   /* iterate over destination port
                                              instances */
    arc = mk-Arc(s, d, arcType, delay);      /* create an arc entity */
    Append(arcs, arc);                       /* enter into arcs table */
   ENDFOR
  ENDFOR
  RETURN(arcs);                              /* return the arcs table */
END
```

--->

```
/* longest delay procedure */

PROCEDURE LongestDelay (IN arcs: <Arc>; nodes: <Node>)

VAR n, n1: Node; p: PortInst; sources: {PortInst}; a: Arc;

BEGIN
 FOR n = Elems(nodes) DO          /* iterate over the nodes */
  p = port(n);                    /* port instance defining the node */
  SET propDelay(n) = 0;           /* initial propagation delay */
  sources = GetSources(arcs, p);  /* get the corresponding source port
                                  instances */
  FOR s = Elems(sources) DO       /* iterate over the source port instances */
   n1 = GetNode(s);               /* get the node corresponding to source
                                  port instance */
   a = GetArc(n1, n);             /* get the arc from node n1 to node n */

   /* check: arc delay + propagation delay of source >
     propagation delay of the node being processed */
   IF ((delay(a) + propDelay(n1)) > propDelay(n))

    /* update propagation delay of node being processed */
    THEN propDelay(n) = delay(a) + propDelay(n1);

   ENDIF
  ENDFOR
 ENDFOR
END
```

Figure 6.7 Timing Analysis: Data Structures and Algorithms

The type *Arc* defines the network arcs each of which connects a source port instance to a destination port instance, has an arc type (wire or transistor), and a corresponding delay. Type *Node* defines the network node in terms of the port instance it represents, and the signal propagation delay from a source port instance to its port instance. Variable *arcs* specifies the table of network arcs; variable *nodes* specifies the sequence of network nodes. The main procedure *CalculateDelay* works by first setting up the arcs table (*SetUpArcs*), sorting the nodes topologically (*TopoSort*), and then computing the longest delay to each node (*LongestDelay*). This algorithm only works for networks with no cycles (i.e.

circuits with no feedback). Function *SetupArcs* sets up the arc table by iterating over the set of wires and transistors in the network (that defines the 4-bit adder), and calculating the arc instance corresponding to each of them. The function *TopoSort* computes the topologically sorted list of network nodes. The procedure *LongestDelay* computes the longest delay to each node using the recursive formula given earlier.

6.6. Summary

In this chapter we have presented an illustrative example VLSI circuit design of a 4-bit adder. The database schema specifications consist of a set of type, class, relationship, and operation specifications relevant to the layout and schematic representations of the 4-bit adder. We also present examples of design database processing involving gate-level logic simulation and timing analysis computations. The processing algorithms are implemented by IDL routines and interface with the database through imported schema specifications and the semantic data model operations. Similar routines can be written to produce structured data files formatted for processing by standard design tools external to the system. The conciseness and conceptual clarity of these specifications illustrate clearly the benefits and advantages of the design database system.

Chapter 7

FORMAL SPECIFICATIONS

7.1. Introduction

Formal specifications of software systems lead to (a) greater understanding of the problem and the appropriate design solution; (b) proofs of correctness of the implementation with respect to the specification; and (c) "cleaner" implementation architectures [CHJ86, Jon86, McG86].

* *Model-based (Operational) specifications* describe system behaviour in terms of a model constructed from abstract and concrete primitives.
* *Algebraic (Axiomatic) specifications* describe system behaviour by defining the relationships among operations in terms of a set of axioms without reference to an explicit model.

In this chapter we describe model-based (VDM) and algebraic (Larch) specification systems. We also discuss the technique of denotational semantics to formally define the semantics of programming languages. We then apply these specification techniques to formally define the following:

* The IDL structured type system
* The IDL Language excluding the embedded semantic data model operations (operationally specified in Section 4.9) and the structured type operators (specified explicitly in Section 7.3.1, axiomatically in Section 7.4.1 and operationally in Section 7.5).

IDL's type system is specified (a) by the VDM type system (Section 7.3.1); (b) axiomatically by the Larch auxiliary specifications (Section 7.4.1); and (c) operationally by the SVIM subsystem implementation (Section 7.5).

VDM is a system for defining model-based (operational) specifications of complex software systems. As with all such tools, it captures only certain aspects of the target system. However, it is still evolving and new concepts and capabilities are being introduced into VDM which are enhancing its modeling power, and consequently, its utility as a specification system. VDM specifications are based on user-specified (abstract) types (obtained by applying a set of "type constructors"), and functions and operations that define system behaviour. The type constructors which define the "structured types" from component types include those of IDL - Set, Sequence, Tuple and Union. Section 7.3.1 contains an explicit specification of these structured types (and their operators) in terms of direct definitions. In addition to providing an operational specification of the IDL type system, these serve to axiomatize VDM on which the denotational semantics specifications of IDL in Section 7.7 are based.

The Larch Auxiliary Specifications of Section 7.4.1 provide an algebraic approach to specifying the data abstractions defined by the IDL type system. Larch is a two-tier specification system designed for program verification. The "auxiliary specifications" define data abstractions (data types), and the "interface specifications" define procedural abstractions. These specifications can be used in proving program implementations correct with respect to specifications of system behaviour.

SVIM (Section 7.5) is a subsystem that implements a part (values and operators) of the IDL type system. It can be used as the basis for a "denotational semantics specifications interpreter" that (a) executes the denotational semantics specifications of IDL thus providing a "definitional interpreter"; and (b) provides a rapid translator prototyping tool. It can also be used as a building block in an implementation of IDL.

The formal semantics of IDL are given as "denotational semantics" specifications. Denotational semantics is a technique for specifying formally the semantics of

programming languages. As described in Section 7.6, this involves defining meaning functions that map the language's syntactic units into well-defined mathematical objects. The meaning function of a syntactic unit is defined in terms of those of its component parts as defined by the language's syntax. These specifications are written in terms of VDM constructs (Section 7.7) and formalize the following parts of IDL: values, types, variables, expressions and basic statements.

7.2. Vienna Development Method

The *Vienna Development Method* (VDM) [BjJ82, Jon86] is a model-based specification system that has been used successfully in a number of industrial applications. VDM was developed at the IBM Vienna Research Laboratories during the 1970s. Since then VDM work has continued in many centres including the Dansk Datamatik Center, Copenhagen, University of Oxford, University of Manchester, and Standard Telecommunications Laboratory in England. VDM has its origins in VDL (Vienna Definition Language) [CHJ86, McG86] which was used to specify an abstract machine interpreter for PL/I thus providing an "operational semantics" for the language. Since then VDM has developed into a general-purpose software development method. It has been applied to the formal specification of the programming language Ada [Oes80], the software engineering environment KAPSE [Dep81], and the system CHILL [CCC81].

Other model-based specification systems include Z [Abr80, Suf82] developed by the Programming Research Group at University of Oxford, HOS [Mar82] developed by Higher Order Software Inc. in Cambridge, Massachussetts, and GIST [Wil82] developed by the Information Sciences Institute, University of Southern California.

7.3. VDM Specifications

A VDM model consists of a set of (abstract) *objects* and a set of *functions* and *operations*. The objects represent *system inputs, system outputs* or a *system state*. The values that can be assigned to these objects are typed by built-in or user-defined (abstract) *types* which characterize their behaviour. Functions and operations define the system behaviour and are specified implicitly (by pre- and post-conditions on system inputs, outputs and state) or defined explicitly (by, possibly recursive, algorithms).

7.3.1. The VDM Type System

The built-in *basic data types* of VDM include Z (integers), N (natural numbers), N_1 (positive integers), R (real numbers) *Bool* (Boolean values) etc. Users can define their own *structured types* by applying *type constructors* to the built-in and other previously defined types. The VDM type constructors are:

Scalar Type Constructors

A *named type* introduces a meaningful name for an existing type. For example the specification:

Distance = R

makes values of type *Distance* to be real numbers.

An *enumerated type* defines a type whose values are specified by enumerating the elements of a set. For example the specification:

Colour = { Red, Green, Yellow, Blue }

defines an enumerated type *Colour* with (abstract) values corresponding to the given colours. For an enumerated type $E = \{ e_1, ..., e_n \}$, the following type operators are also defined:

* *ord* returns the ordinal number for a given value in the type:

ord: E $\rightarrow N_1$
ord(e) = n where n is the ordinal number of value e in the list $(e_1, ..., e_n)$

A *subrange type* defines a type whose values are subranges of built-in numeric or previously defined enumerated types. For example the specification:

Hour = { 1:12 }

defines the domain of type *Hour* to be integer values in the range 1 to 12.

Set Type Constructor

The set type constructor defines a type whose values are finite sets. A set type *S* with constituent type *T* is defined by:

S = **set of** T

S is now the "finite powerset" and the values of type *S* are now finite sets of values of type *T*. The following *type operators* are also defined for type *S*. The structured type operators are specified by *direct definitions* (i.e. procedurally) in terms of standard set-theoretic operations. These structured type operators are modeled by *functions* (which operate on *values*) unlike *operations* (which modify a *state* (global named value). An actual implementation of these operators may, however, use the state information depending on the properties associated with the type (i.e. mutability vs. immutability and pointer vs. value semantics).

* *emptySet* creates an empty set-value:

  emptySet: {**empty-string**} $\rightarrow$ S
  emptySet(**empty-string**) Δ { }

  The symbol Δ above is to be read as "is defined as".

* *insert* inserts a value of the element type into a set value:

  insert: S $\times$ T $\rightarrow$ S
  insert(s, t) Δ s $\cup$ {t}

* *delete* deletes a value of the element type from a set value:

delete: $S \times T \rightarrow S$
delete(s, t) Δ s - {t}

* *size* returns the cardinality of the set value:

size: $S \rightarrow N$
size(s) Δ n where n is the cardinality of the set s

* *isEmpty* tests if the set is empty:

isEmpty: $S \rightarrow$ Bool
isEmpty(s) Δ TRUE if s is empty
FALSE otherwise

* *member* tests if a specified element is in the set:

member: $S \times T \rightarrow$ Bool
member(s, t) Δ TRUE if $t \in s$
FALSE otherwise

* *choose* selects an arbitrary element in the set:

choose: $S \rightarrow T$
choose(s) Δ t where $s \neq \{\}$ and $t \in s$

Here, *choose* is a "non-deterministic" function in the sense that the value returned by it is determined, in addition to the value of *s*, by the hidden state of its representation. This hidden state can, however, be indirectly determined by using a sequence of *choose* and *rest* (defined below) operations that results in a sequence of elements in the set.

* *rest* returns the set with the element selected by the immediately preceding choose deleted:

rest: $S \rightarrow S$
rest(s) Δ delete(s, choose(s))

* *equal* is a predicate that tests for equality of two set values:

equal: $S \times S \rightarrow$ Bool
equal(s_1, s_2) Δ $s_1 \subseteq s_2$ & $s_2 \supseteq s_1$

Reporting errors for type operators

Error checking and reporting are built into a higher-level layer that provides procedures and functions with appropriate error code or message parameters. In the case of the set *insert* and *delete* operators, for example, there could be corresponding user-defined operations *Insert* and *Delete* that make use of the set operator *member* to make a check before inserting or deleting:

```
PROCEDURE Insert (IN s: Set; e: Element; OUT e-message: STRING)
BEGIN
 e-message = "";
 IF member(s, e) THEN e-message = "element is already a member!";
          ELSE insert(s, e);
END

PROCEDURE Delete (IN s: Set; e: Element; OUT e-message: STRING)
BEGIN
 e-message = "";
 IF member(s, e) THEN delete(s, e); ELSE e-message = "element not in the set";
END
```

If the check fails, these operations return a meaningful message to the calling routine through the standard parameter passing mechanism. The *Insert* and *Delete* operations with the appropriate message parameter could be provided in a utility library.

Sequence Type Constructor

The sequence type constructor defines a type whose values are sequences of values of the constituent type. A sequence type *Q* with element type *T* is defined by:

Q = seq of T

The values of type *Q* are finite sequences of values of type *T*. The following type operators are also provided:

* *emptySeq* creates an empty sequence:

emptySeq: {**empty-string**} → Q
emptySeq(**empty-string**) Δ <>

* *length* returns the length of the sequence:

length: Q → N
length(q) Δ n where n is the length of the sequence

* *insert* inserts an element value at a specified position in the sequence:

insert: Q × T × N_1 → Q
insert(q, t, n) Δ <q'[i] | for 1≤i<n q'[i] = q[i]; for i=n q'[i] = t;
for n<i≤length(q)+1 q'[i] = q[i-1]

* *delete* deletes the element at a specified position in the sequence:

delete: Q × N_1 → Q
delete(q, n) Δ <q'[i] | for i≤i<n q'[i] = q[i]; for n<i+1≤length(q) q'[i] = q[i+1]>

* *append* appends an element to the sequence:

append: Q × T → Q
append(q, t) Δ <q'[i] | for 1≤i≤length(q) q'[i] = q[i]; q'[length(q)+1] = t>

* *select* returns the element in the specified position in the sequence:

select: Q × N_1 → T
select(q, n) Δ q[n] where n ≤ length(q)

* *isEmpty* tests if the sequence is empty:

isEmpty: Q → Bool
isEmpty(q) Δ TRUE if q is empty
FALSE otherwise

* *member* tests if a specified element is in the sequence:

member: Q × T → Bool
member(q, t) Δ TRUE if ∃i · 1 ≤ i ≤ length(q) & q[i]=t
FALSE otherwise

* *head* returns the first element in the sequence:

head: Q → T
head(q) Δ q[1]

* *tail* returns the sequence with the first element deleted:

$$\text{tail: } Q \rightarrow Q$$
$$\text{tail}(q) \Delta \text{ delete}(q, 1)$$

* *equal* is a predicate that tests for equality of two sequences:

$$\text{equal: } Q \times Q \rightarrow \text{Bool}$$
$$\text{equal}(q_1, q_2) \Delta \text{ length}(q_1)=\text{length}(q_2) \; \& \; \forall i \cdot 1 \leq i \leq \text{length}(q_1) \cdot \text{select}(q_1, i)=\text{select}(q_2, i)$$

Union Type Constructor

The union type constructor defines a type whose values are "tagged" values of the constituent types. A union type U with constituent types $T_1, ..., T_n$ is defined by:

$$U = T_1 \mid .... \mid T_n$$

The domain of U is now the disjoint union of the domains of $T_1, ..., T_n$. The following operators are also defined:

* *inject* injects a value of a constituent type into a value of the union type:

$$\text{inject-}i : T_i \rightarrow U$$
$$\text{inject-}i\,(t_i) \Delta (T_i, t_i) \text{ if } t_i \in T_i$$
$$\text{undefined otherwise}$$

The tag T_i denotes that the value is from the corresponding type.

* *inspect* tests if a value of the union type belongs to a specified constituent type:

$$\text{inspect-}i : U \rightarrow \text{Bool}$$
$$\text{inspect-}i\,(u) \Delta \text{ TRUE if u has tag } T_i$$
$$\text{FALSE otherwise}$$

* *project* projects a value of the union type onto the corresponding value of a constituent type:

$$\text{project-}i : U \rightarrow T_i$$
$$\text{project-}i\,(T_i, t_i) \Delta t_i \text{ if } t_i \in T_i$$
$$\text{undefined otherwise}$$

There is a family of *inject*, *inspect* and *project* operators for each union type.

* *equal* is a predicate that tests for equality of two union values:

$$\text{equal: } U \times U \rightarrow \text{Bool}$$
$$\text{equal}(u_1, u_2) \Delta \exists i \cdot \text{inspect-i}(u_1)=\text{inspect-i}(u_2)=\text{TRUE} \;\&\; \text{equal}(\text{project-i}(u_1), \text{project-i}(u_2))$$

Map Type Constructor

The map type constructor defines a type whose values are *maps* (finite total functions) from a specified *domain* to a specified *codomain*. A map type M with domain D and codomain R is defined by:

$$M = \textbf{map } D \textbf{ to } R$$

The values of type M are mappings from domain of D to domain of R. A value of type M is constructed by giving a set of pairs of corresponding values of types D and R:

$$m: M \text{ and } m = \{ d_1 \rightarrow r_1, ..., d_n \rightarrow r_n \} \text{ such that } d_i \neq d_j \text{ for } i \neq j$$

The following operators are also defined;

* *domain* returns the finite set of values in the mapping's domain:

$$\text{dom: } M \rightarrow \text{set of } D$$
$$\text{dom}(m) \Delta \{ d \mid (d \rightarrow r) \in m \}$$

* *range* returns the finite set of values in the mapping's codomain:

$$\text{rng: } M \rightarrow \text{set of } R$$
$$\text{rng}(m) \Delta \{ r \mid (d \rightarrow r) \in m \}$$

* *map application* returns a value of codomain type corresponding to the given value of the domain type:

$$_(_): M \times D \rightarrow R$$
$$m(d) \Delta r \quad \text{where } (d \rightarrow r) \in m$$

* *map composition* returns a new map which is the composition of the given maps:

M_1 = map D to R'
M_2 = map R' to R
M_3 = map D to R
$_o_: M_1 \times M_2 \rightarrow M_3$
$m_1 \circ m_2 \Delta \{ d \rightarrow r \mid \exists r' \in R' (d \rightarrow r') \in m_1 \& (r' \rightarrow r) \in m_2 \}$

* *merge* merges two given maps:

$_\cup_: M \times M \rightarrow M$

For m_1 and m_2 such that $dom(m_1) \cap dom(m_2) = \phi$,
$m_1 \cup m_2 \Delta \{ d \rightarrow r \mid ((d \rightarrow r) \in m_1$ OR $(d \rightarrow r) \in m_2) \}$

* *override* merges the two mappings overriding the mappings of the first map by the mappings of second map for common domain values:

$_+_: M \times M \rightarrow M$
$m_1 + m_2 \Delta \{ d \rightarrow r \mid ((d \rightarrow r) \in m_1 \& \neg (d \in dom(m_2))$ or $(d \rightarrow r) \in m_2) \}$

* *domain deletion* returns a new mapping whose domain is the domain of the given mapping restricted to those elements not in the given set:

$_/_: M \times$ set of $D \rightarrow M$
$m / s \Delta \{ d \rightarrow m(d) \mid d \in (dom(m) - s) \}$

* *domain restriction* returns a new mapping whose domain is restricted to those elements in the given set:

$_|_: M \times$ set of $D \rightarrow M$
$m | s \Delta \{ d \rightarrow m(d) \mid d \in (dom(m) \cap s) \}$

Composite Type Constructor

The composite type constructor (::) defines a type whose values represent "composite objects" with named "component objects". Such a type can be used to model records of Pascal, structures of C, PL/I etc. A composite type C with fields f_1, ..., f_n of types respectively T_1, ..., T_n can be defined by:

$C :: f_1 : T_1$
....
....
$f_n : T_n$

The values of type C are tuples with attributes $f_1, ..., f_n$ and corresponding values of types $T_1, ..., T_n$. The following composite type operators are also defined:

* *make function* constructs a value of the composite type from given field values:

$$\text{mk-}C : T_1 \times .... \times T_n \rightarrow C$$
$$\text{mk-C}(t_1, ..., t_n) \Delta (f_1 = t_1, ..., f_n = t_n)$$

* *selector function* retrieves a component value corresponding to the given field from the composite value:

$$f_i : C \rightarrow T_i$$
$$f_i(f_1 = t_1, ..., f_n = t_n) \Delta t_i$$

* μ-*function* changes the value assigned to a specified field of a composite value:

$$\mu_i : C \times T_i \rightarrow C$$
$$\mu_i((f_1 = t_1, ..., f_n = t_n), t_i') \Delta (f_1 = t_1, ..., f_i = t_i', ..., f_n = t_n)$$

There is a family of selector and μ functions for each composite type. The μ functions can be used in combination to change the values of multiple fields in a single operation.

* *equal* is a predicate that tests for equality of two composite values:

$$\text{equal: } C \times C \rightarrow \text{Bool}$$
$$\text{equal}(c_1, c_2) \Delta \forall i \cdot 1 \leq i \leq n \cdot \text{equal}(f_i(c_1), f_i(c_2))$$

Composite types can also be used to specify recursively constructed objects. For example, a *binary search tree* can be defined by the specifications:

Bst = [Node]
[Node] = Node ∪ {**nil**}
Node :: left: Bst
key: N
right: Bst

where **nil** is a special object that denotes the absence of a *Node* object. Here, *Bst* is the

abstract binary search tree type being defined; *Node* is the type of each node in the tree. Each node stores a key value (*key*) and two (possibly *nil*) binary search subtrees (*left*, *right*). A specific binary search tree is defined by its root node (a value of type *Node*). The search tree property requires that for every node in the tree all keys in a right branch are greater than the key in the node which in turn is greater than all the keys in the left branch. This property is specified by the binary search tree node invariant which can be defined as:

```
inv-Node(mk-Node(lst, k, rst)) Δ
  (∀lk ∈ retrns(lst) · lk < k) & (∀rk ∈ retrns(rst) · rk > k)
```

where the function *retrns* is defined as:

```
retrns: Bst → set of N
retrns(b) Δ cases b of
            nil → {}
            mk-Node(lst, k, rst) → retrns(lst) ∪ {k} ∪ retrns(rst)
          end
```

Valid values of recursive types are constructed in a finite sequence of steps by combining primitive or already constructed values.

Users can define abstract types (*A*) in terms of *representation types* (*R*) constructed by using the type constructors, and setting up a mapping between abstract values and corresponding representation values. Such a mapping is defined in terms of the *retrieve function* (*Retrv*) specified as:

```
Retrv: R → A
```

The mapping cardinality of *Retrv* from *R* to *A* is many-to-one and reflects the fact that many different representation values may denote a single abstract value. Thus the representation value is more "detailed" and the retrieve function may be thought of as regaining the abstraction from the (implementation) details. To ensure that all representation values that may arise are "legal", i.e. they can be mapped to an abstract value by the retrieve function, a *representation invariant* predicate which constrains the domain of

representation type is defined. The retrieve function is then a total function. The other notion associated with representation and abstract types is that of *adequacy*. A representation type is adequate if there is at least one representation value corresponding to any abstract value. This requirement can be satisfied by discharging adequacy proof obligation:

$$\forall a \in A \; \exists r \in R \cdot (Retrv(r) = a)$$

7.3.2. A VDM Model

A VDM model consists of the following components:

* A collection of *types* (built-in and user-defined abstract types specified in terms of representation types constructed with type constructors, retrieve functions, and representation invariants).
* A collection of *variables* (of specified types) that denote system inputs, outputs and states.
* A collection of *initial states* (in which the state invariant holds).
* A collection of *functions*.
* A collection of *operations*.

The functions and operations define the behavioural aspects of the system. Functions are (implicitly) specified by stating the result to be computed, or (directly) defined by giving an algorithm to compute the desired result. Implicit specifications are declarative, shorter and clearer, and usually define a range of acceptable results. Direct definitions are procedural and usually define a specific result. A template for a function specification can be written as shown below:

f (d: D) r: R
pre-f: .... d ....
post-f: .... d ... r ...

The first line of the specification, called the *signature*, specifies the name of the function, the names and types of its argument and result parameters. The second line specifies the precondition which defines constraint on or properties to be satisfied by the argument values. The third line specifies the postcondition which defines constraints on or properties to be satisfied by the parameter and result values. A function is implemented by giving a direct (algorithmic) definition. If algorithm F implements a function f, then F is correct with respect to f if the following logic formula holds:

$$\forall d \in D \cdot \text{pre-f}(d) \Rightarrow F(d) \in R \ \& \ \text{post-f}(d, F(d))$$

A function defines a behaviour that depends only on its arguments while an operation defines behaviour that depends, in addition to its arguments, also on the *state* of the system. An operation thus requires access to a collection of "external variables" that define the system state. A template for an operation specification can be written as shown below:

OP (a: T_a) r: T_r
ext: rd v_1: T_1; wr v_2: T_2
pre-OP: ... a ... v_1 ... v_2
post-OP: ... a ... v_1 ... v_2' ... v_2 ... r ...

The first line of the specification specifies the name of the operation, and the names and types of its argument and result parameters. The second line specifies external (system state) variables accessible to the operation, and the type of access: read (rd) or read and write (wr). The precondition defines constraints on or properties to be specified by the argument and state values. The post condition defines constraints on or properties to be satisfied by the argument, result and state values. The primed and unprimed versions of the state variable v_2 in the postcondition indicate its values before and after the execution of the operation. Operations can be implemented by programs, procedures or even

individual statements in an implementation language, and such implementations can be proved correct with respect to the operation specifications by using the proof axioms of the language in a manner similar to that for functions.

7.4. Algebraic Specifications

Algebraic (or *axiomatic*) *specifications* define the behaviour of systems in terms of the relationships among a set of operations of the system. Such specifications consist of:

* *Sorts* which are names for sets of values (analogous to data type domains).
* *Operators* which model system operations. These are specified by their functional types (signatures) defined as mappings from a domain to a codomain defined as Cartesian products of lists of specified sorts.
* *Axioms* which are equations that define the relationships among the operators.

Loose (or initial) specifications define a theory (of first-order logic) that consists of the closure of the set of equations derivable from the axioms under the rules of equational inference. *Tight (or final) specifications* define a theory that consists of a "tight closure" of the set of equations derivable from the axioms under the rules: (1) equational inference; (2) structural induction; and (3) the rule that if equality is not derivable under (1) and (2), then inequality holds.

7.4.1. Larch Auxiliary Specifications of the IDL Structured Types

The Larch specification system [GHW85] is based on the algebraic specification technique. Larch's design philosophy is to provide a modular, two-tiered approach to formal specifications that can be useful in program development and verification environments. Larch Auxiliary Specifications are defined in terms of units called *traits*. A trait models a well-defined data abstraction although it need not correspond to an abstract data

type. The trait is defined in terms of *operators* (syntax) and *axioms* (semantics). An operator defines a mapping from its domain to its range; the domain and range are specified in terms of *sorts*. A sort is a set of values and corresponds to the domain of a data type. The axioms of the trait are written as *equations* composed of well-formed *terms*. A term is recursively defined as a constant value of a specified sort, a variable that ranges over the values of a specified sort, or the result of an operator application to its arguments which are terms themselves. The *theory* associated with a trait consists of all the axioms, the inequation ¬ (TRUE = FALSE), and all theorems derivable from these under the axioms and rules of inference of first order predicate calculus with equality. The specification of a trait may also include references to other predefined traits. This allows traits to be combined in well-defined ways to build up abstractions in a hierarchical and modular fashion. An *imported* trait ensures that the theory being defined by the importing trait is a conservative extension of the theory associated with the imported trait; that is, the operators of the imported trait may not be further constrained by the importing trait or any other imported trait. An *included* trait, on the other hand, allows its operators to be further constrained by the equations of the including trait. An *assumed* trait makes its associated operators available in the trait being defined. Trait specifications also include additional clauses which essentially enrich the associated theory. Thus, the *generated by* clause says that all values of the distinguished sort of the trait can be generated by the operators in that clause. The *partitioned by* clause specifies that two unequal terms of the distinguished sort can be distinguished by using only the operators in that clause. The *converts* clause says that the trait's specification adequately defines the collection of operators in that clause. And finally, the *exempts* clause exempts the operators in it from appearing in any constraint equation. A detailed description of Larch can be found in [GHW85].

The Set Type:

We define the set type in terms of trait *SetS* whose sort *S* represents its domain. The associated operators *emptySet, insert, isEmpty, size, member* and *delete* are all inherited from the assumed or imported traits and have the same semantics as in their respective defining traits. Operator *choose* selects some element (unspecified but always the same for a given set value) from the set. Operator *rest* returns the set after deleting the element selected by the immediately preceding *choose*.

```
SetS: trait
 imports SetBasics with [ S for C]
 includes IsEmpty with [ S for C ],
          Member with [ member for ∈ ],
          Size with [ S for C ],
          Container with [ S for C, emptySet for new ],
          Rest with [ S for C ].
 introduces
    choose: S → E    /* E is in the Container trait */
 constrains member, isEmpty, choose, rest, size
  so that for all [ s1, s2, s: S, e: E ]
   (choose(rest(s)) = choose(s)) = FALSE
   isEmpty(rest(s)) = if (size(rest(s)) = 0) then TRUE else FALSE
   size(rest(s)) = if (isEmpty(s)) then 0 else size(s)-1
   member(rest(s), e) = if (choose(s)=e) then FALSE else member(s, e)
 generated by [ emptySet, insert ]
 partitioned by [ choose, member, size, rest ]
 converts [ choose, rest ]
 exempts for all [ s1, s2: S, e: E ] choose(emptySet()), choose(insert(s, e)),
    choose(delete(s, e))
```

The Sequence Type:

We define the sequence type in terms of trait *SeqQ* whose sort *Q* represents its domain. Operators *emptySeq, isEmpty, member, length* and *select* are inherited from the assumed or imported traits. Operator *insert* inserts an element into the n^{th} position of the sequence, operator *delete* deletes the n^{th} element from the sequence, operator *append* appends the given element as the last element of the sequence.

```
SeqQ: trait
 assumes Cardinal
 includes IsEmpty with [ Q for C ],
          Member with [ member for ∈ ],
          Size with [ Q for C, length for size ],
          Index with [ Q for C, select for #[#] ],
          Container with [ Q for C, emptySeq for new ]
 introduces
    append: Q × E → Q            /* E is in the Container trait */
    insert: Q × E × Card → Q
    delete: Q × Card → Q
 constrains delete, insert, append, isEmpty, member, length, select
   so that for all [ q: Q, e, e1, e2: E, n, n1, n2: Card ]
    isEmpty(append(q, e)) = FALSE
    isEmpty(insert(q, e, n)) = FALSE   for n ≤ length(q)
    isEmpty(delete(q, n)) = if length(delete(q,n))=0 then TRUE else FALSE
    member(append(q, e1), e2) = if ¬(e1=e2) then member(q, e2) else TRUE
    member(insert(q, e1, n), e2) = if ¬(e1=e2) then member(q, e2) else TRUE
    length(append(q, e)) = length(q)+1
    length(insert(q, e, n)) = length(q)+1  for n ≤ length(q)
    select(append(q, e), n) = if n≤length(q) then select(q, n) else if n=length(q)+1 then e
    select(insert(q, e, n1), n2) = if n2<n1 then select(q, n2) else
                                   if n2>n1 then select(q, n2-1) else e
    select(delete(q, e, n1), n2) = if n2<n1 then select(q, n2) else
                                   if n2>n1 then select(q, n2+1) else select(q, n1)
 generated by [ emptySeq, append ]
 partitioned by [ select, length ]
 converts [ delete, insert, append ]
 exempts for all [ n: Card, e: E, q: Q ] delete(emptySeq(), n), member(delete(q, n), e),
  length(delete(q, e))
```

The Composite Object Type:

We define the composite object type in terms of trait *CompC* whose sort *C* represents its domain. $f_1, ..., f_n$ are fields and $T_1, ..., T_n$ are sorts of the traits corresponding to the field types.

CompC: **trait**
includes Container **with** [T_i **for** C] (i = 1, ..., n)
introduces
mk-C: $T_1 \times ... \times T_n \rightarrow$ T
f_i: T $\rightarrow T_i$ (i = 1, ..., n)
μ_i: T $\times T_i \rightarrow$ T (i = 1, ..., n)
constrains so that for all [t: T, t_i: T_i]
f_i(mk-C(t_1, ..., t_n)) = t_i
f_i(μ_i(mk-C(t_1, ..., t_n), t_i')) = t_i'

generated by [mk-T]
partitioned by [f_i] (i = 1, ..., n)
converts [mk-C, f_i, μ_i] (i = 1, ..., n)

The Discriminated Union Type:

We define the discriminated union type in terms of trait *UnionU* whose sort *U* represents its domain. T_1, ..., T_n are the sorts of traits corresponding to the constituent types.

UnionU: **trait**
includes Container **with** [T_i for C] (i = 1, ..., n)
introduces
inject-i: $T_i \rightarrow$ U (i = 1, ..., n)
inspect-i: U $\rightarrow$ Bool (i = 1, ..., n)
project-i: U $\rightarrow T_i$ (i = 1, ..., n)
constrains inject-i, inspect-i, project-i **so that for all** [u: U, t_i: T_i]
inspect-i(inject-i(t_i)) = TRUE
project-i(inject-i(t_i)) = t_i
generated by [inject-i] for i = 1, ..., n
partitioned by [inspect-i, project-i] for i = 1, ..., n
converts [inject-i, inspect-i, project-i] for i = 1, ..., n

7.5. SVIM: An Implementation of the Values and Operators of the IDL Structured Types

The SVIM (Structured Value Implementation Module) subsystem implements the values and operators of the IDL structured types defined in Section 5.9.2. This subsystem can serve as the basis for implementing a *denotational semantics specification interpreter* thus providing a formal *operational semantics* for the database system. The specification interpreter would also serve as a rapid compiler prototyping tool. Also, the SVIM subsystem can be used as a building block in an implementation of the IDL type system.

An interactive command interface to SVIM has also been implemented. Using this interface, a user can interactively create structured values and operate on them using the corresponding structured type operators. The system functions by parsing input commands and invoking appropriate SVIM routines that actually implement the values and operators. The command language is explained below. The syntactic conventions are: keywords are in roman font and user supplied parameters are italicized; blanks separate the command line into tokens except inside string values enclosed in double quotes. The user supplied parameters are as follows:

type is a basic data type from the set { Int, Real, Bool, String }.

value is a basic data type value.

valueID is an integer that represents the internal, system-generated identifier assigned to each value created by the user.

setID is the *valueID* of a set value.

seqID is the *valueID* of a sequence value.

tupleID is the *valueID* of a tuple value.

unionID is the *valueID* of a union value.

position is an integer denoting a position in a sequence.

attribute is an alphanumeric string that denotes an attribute name in a tuple value.

tag_string is an alphanumeric string that denotes the tag of a union value.

(1) store *type value*

stores a value *value* of an atomic data type *type* in the value table.

(2) remove *type value*

removes a value *value* of an atomic data type *type* from the value table.

(3) lookup *type value*

looks up a value *value* of an atomic data type *type* in the value table.

Set Type Operators

(4) create empty-set

create { }

create an empty set.

(5) insert *type value* into **setID*

insert **valueID* into **setID*

The first form inserts a value *value* of an atomic data type *type* into set with value identifier *setID*; the second form inserts a value with identifier *valueID* into the set with identifier *setID*.

(6) delete *type value* from **setID*

delete **valueID* from **setID*

The first form deletes a value *value* of an atomic data type *type* from the set with value identifier *setID*; the second form deletes a value with identifier *valueID* from the set with identifier *setID*.

(7) member *type value* in **setID*

member **valueID* in **setID*

The first form checks if a value *value* of an atomic data type *type* is in the set with value identifier *setID*; the second form checks if a value with identifier *valueID* is in the set with identifier *setID*.

(8) size **setID*

returns the size of the set with value identifier *setID*.

(9) choose **setID*

returns an arbitrary member of set with value identifier *setID*.

Sequence Type Operators

(10) create empty-seq

create <>

create an empty sequence.

(11) append *type value* to **seqID*

append **valueID* to **seqID*

The first form appends a value *value* of a basic data type *type* to the sequence with the value identifier *seqID*; the second form appends a value with identifier *valueID* to the sequence with identifier *seqID*.

(12) insert-seq *type value* into **seqID* at *position*

insert-seq **valueID* into **seqID* at *position*

The first form inserts a value *value* of a basic data type *type* into the sequence with the value identifier *seqID* at position *position*; the second form inserts a value with identifier *valueID* into the sequence with identifier *seqID* at position *position*.

(13) delete-seq *position* from **seqID*

deletes the element at position *position* from the sequence.

(14) member-seq *type value* in **seqID*

member-seq **valueID* in **seqID*

The first form checks if a value *value* of a basic data type *type* is in the sequence with value identifier *seqID*; the second form checks if a value with identifier *valueID* is in the sequence with identifier *seqID*.

(15) length **seqID*

returns the length of the sequence with value identifier *seqID*.

(16) select *position* in **seqID*

returns the element at position *position* in the sequence with value identifier *seqID*.

Tuple Type Operators

(17) create tuple ($attr_1 = value\text{-}spec_1, ..., attr_n = value\text{-}spec_n$)

where $attr_i$ is alphanumeric string that begins with a letter, and $value\text{-}spec_i$ has the form *type value* (value *value* of basic data type *type*) or **valueID* (value with value identifier *valueID*). This command creates a tuple with attributes $attr_1$ through $attr_n$ and corresponding values $value\text{-}spec_1$ through $value\text{-}spec_n$.

(18) project **tupleID* on *attr*

projects the tuple with identifier *tupleID* on its attribute *attr*.

Union Type Operators

(19) create union (tag = *tag*, value = *value-spec*)

creates a union value with tag *tag* and value *value-spec*.

(20) tag **unionID*

returns the tag component of the union value with the identifier *unionID*.

(21) value **unionID*

returns the value component of the union value with the identifier *unionID*.

Value Interpretation

(22) interpret **valueID*

interpret-rec **valueID*

The first form prints out the value identifier and type of the value with the identifier *valueID*; if the value is a structured value, the identifiers and types of all its components are printed. The second form interprets recursively on the components; the recursion terminates after all the atomic data type value components are printed out.

(23) list

lists all the identifiers and types of all values stored in the value table.

(24) quit

terminates the session.

7.6. Denotational Semantics

Denotational semantics is a formal specification technique for defining the semantics of programming languages [BjJ82, Gor79, McG80, Ten76]. A denotational semantics specification is given in terms of a *meaning function* specified as:

$$M: L \rightarrow DEN$$

where domain *L* is the set of all syntactic objects of the language being defined and domain *DEN* is the set of all "denotations" which are well-defined mathematical objects like integers, reals, sets, functions etc. The meaning function maps syntactic objects into corresponding abstract mathematical (or semantic) objects.

Typically, the objects in *L* are "structured hierarchically" and are built up from component objects as defined by the *abstract syntax* of the language. Typical syntactic categories defining the syntax of programming languages include *variables, expressions, statements, control structures, modules, programs* etc. The denotational semantics of

such programming languages are then defined by a hierarchy of meaning subfunctions corresponding to each syntactic category. The semantics of a structured object is then defined in terms of the semantics of the component objects. For example, if binary numerals (in domain *BinNml*) have a syntactic structure defined by the abstract syntax:

BinNml = 0 | 1 | BinNml 0 | BinNml 1

the semantics of binary numerals can be defined by mapping them into their corresponding integer values:

M: BinNml → Z
M[b] Δ case b of
  0 → 0
  1 → 1
  b_1 0 → 2 * M[b_1]
  b_1 1 → 2 * M[b_1] + 1
  end

The meaning function, which maps a binary numeral into the integer value it represents is defined by case analysis of its structure. As another example, the semantics of arithmetic expressions can be defined in terms of the semantics of their constituent subexpressions and the arithmetic operators:

M: ArithmExpr → Number
M[x+y] Δ M[x] + M[y]

where *ArithmExpr* and *Number* are the domains respectively of arithmetic expressions and numbers. The semantics of an expression involving the sum of two terms x and y is given as the sum of the semantics of the two terms.

The denotations (in *DEN*) may be quite complex and may involve recursively defined objects. Hence it is necessary to ensure that the denotational domains are well-defined and do not involve arbitrary sets or functions which may give rise inconsistencies or paradoxes of naive set theory. This is achieved by restricting these domains to the so called *Scott domains* which are complete lattices with a partial order relation that is interpreted as an *approximation relation* and two special objects *bottom* (undetermined object

that approximates all other objects) and *top* (overdetermined or inconsistent object that is approximated by all other objects). The set of *primitive domains* includes *integers, reals, Booleans* etc. with each set augmented with the top and bottom elements. More complex domains are constructed by applying *domain constructors* to specified domains. The domain constructors include *sum, discriminated union, Cartesian product* and *continuos function (mapping)*. The VDM types and type constructors defined in Section 6.3.2 correspond to these domains and domain constructors. It has been shown that all domains specified by these constructors are well-defined, permit finite approximations (of infinite objects), and can model recursively defined objects with recursively defined (or *reflexive*) domains. Reflexive domains can include function spaces defined in terms of mappings (in turn, modeled by sets, pairs, pointers etc.)

As an example consider the semantics associated with the notion of variables in a programming language. Assuming a simple language in which variables are implemented as store locations, the meaning of variables can be defined in terms of an *environment* (Env) which maps *variables* (Var) to *locations* (Loc), and a *store* (Store) which maps locations to *scalar values* (ScalarValue):

Env = **map** Var **to** Loc
Store = **map** Loc **to** ScalarValue
M: Var → (Env → (Store → ScalarValue))
M[v](e)(s) Δ s(e(v))

where the meaning function *M* maps variable *v* in environment *e* to the value assigned to it by store *s*.

To summarize, a denotational semantics specification of a language involves the following:

* A collection of *syntactic domains* and syntax rules (*abstract syntax*) which define how the syntactic domains are constructed.

* A collection of *semantic domains* (denotations) and their definitions in terms of primitive domains and domain constructors.
* A collection of *meaning functions* which map syntactic objects into corresponding semantic objects.

The advantages of denotational semantic specifications of programming languages are (a) every syntactic object is given a unique meaning by being mapped on to a specified denotation, and hence there is no ambiguity, inconsistency or incompleteness associated with informal descriptions of language semantics; (b) the meaning functions and denotations are mathematical objects and formal mathematical techniques can be used to prove results about their properties; and (c) a *denotational specification interpreter* can be used to compile and execute the specifications of particular programs in the language thus providing a "definitional interpreter" for the language.

7.7. A Denotational Semantics for IDL

In this section, an outline of a denotational definition of the semantic data model and the integrated database language is sketched. The denotational semantics of the structured type operators and the data model operations embedded in the language is not given as these are operationally specified in Sections 4.9 and 7.3.1.

7.7.1. Abstract Syntax

The syntactic categories used in defining the abstract syntax are as follows:

An IDL program consists of an initialization statement, a body composed of a sequence of "blocks" and a close-database statement:

```
Program :: init: InitStmnt
           body: seq of Block
           final: CloseDB
```

An initialization statement is a define-database or an import-database statement:

InitStmnt = DefDB | ImportDB

A define-database statement has a database name and a database file directory:

DefDB :: name: DBName
dir: Directory

A database name is an identifier (domain *Id*) and a directory is assumed to be a a file directory (a predefined domain *Directory*). An import-database statement has the same structure as a define database except that its effect is to set up an initialized environment by reading specified files already created in the given directory:

ImportDB :: name: DBName
dir: Directory

A close-database statement specifies the database name:

CloseDB = DBName

A block is a declaration or statement:

Block :: Decl | Stmnt

A declaration can be a user type declaration, a variable declaration, a class declaration, a procedure declaration, or a function declaration:

Decl = TypeDecl | VarDecl | ClassDecl | ProcDecl | FunDecl

A user type declaration is either an enumerated type or a tuple type:

TypeDecl = EnumeratedType | TupleType

An enumerated type has a type name and a set of basic data type values:

EnumeratedType :: name: TypeName
values: set of BasicTypeValue

A tuple type has a type name and a set of attribute specifications:

TupleType :: name: TypeName
attribs: set of AttribSpec

An attribute specification has an attribute name and a corresponding domain:

AttribSpec :: name: AttribName
dom: DomSpec

An attribute domain specification is a basic data type, a class name, or an enumerated type:

DomSpec = BasicType | ClassName | EnumeratedTypeName

A Variable declaration specifies a variable name and its type:

VarDecl :: name: VarName
type: Type

A variable name is an identifier. A variable type can be a basic type, a set type, a sequence type, a class name (the *Entity* type), an enumerated type or a tuple type:

Type = BasicType | SetType | SeqType | ClassName |
EnumeratedTypeName | TupleTypeName
BasicType = { **Int, Real, Bool, String** }
SetType = BasicType | ClassName | EnumeratedTypeName | TupleTypeName
SeqType = BasicType | ClassName | EnumeratedTypeName | TupleTypeName

A class declaration consists of a class name, a set of attributes, a set of constraints, a set of superclasses and a set of subclasses:

ClassDecl :: name: ClassName
attrib: set of Attrib
constr: set of Constr
super: set of ClassName
sub: set of ClassName

A class name is an identifier. An attribute has a name and a domain which is a type:

Attrib :: name: AttribName
dom: Type

An attribute name is an identifier. An attribute with a *ClassName* domain has the system-defined abstract data type *Entity*. *super* and *sub* are immediate super and subclasses; *Is-a** is the least quasi-order on these relationship pairs. A procedure declaration has a name, a sequence of input and output parameters, a set of local types, a set of local variables, and a body composed of a sequence of statements:

```
ProcDecl :: name: ProcName
            inParams: seq of VarDecl
            outParams: seq of VarDecl
            localTypes: set of TypeDecl
            localVars: set of VarDecl
            body: seq of Stmnt
```

A function declaration has a name, a sequence of input parameters, a result type, a set of local types, a set of local variables, and a body composed of a sequence of statements:

```
FunDecl :: name: FunName
           params: seq of VarDecl
           resultType: Type
           localTypes: set of TypeDecl
           localVars: set of VarDecl
           body: seq of Stmnt
```

A statement is an assignment, a procedure invocation, an if-then-else construct, an if-then construct, a while-do loop, a for-do loop, or a data model operation that executes a command:

```
Stmnt = Assgn | ProcInvoc | IfThenElse | IfThen | WhileDo | ForDo | DataModelOp
```

An assignment consists a target variable and a value expression:

```
Assgn :: lhs: VarName
         rhs: Expression
```

A procedure invocation consists of the procedure name, a sequence of input arguments, and a sequence of output arguments:

```
ProcInvoc :: name: ProcName
             inputArgs: seq of Expression
             outputArgs: seq of VarName
```

The for-do iteration loop consists of a loop variable, an iterator, and a body composed of a sequence of statements:

```
ForDo :: loopVar: VarName
         iterator: Iterator
         body: seq of Stmnt
```

An if-then-else construct consists of a condition part, a then part and an else part.

```
IfThenElse ::   cond: BoolExpr
                then: seq of Stmnt
                else: seq of Stmnt
```

An if-then construct consists of a condition part, and a then part:

```
IfThen ::   cond: BoolExpr
            then: seq of Stmnt
```

A while-do loop consists of a condition part and a body composed of a sequence of statements:

```
WhileDo :: cond: BoolExpr
           body: seq of Stmnt
```

An expression can be an arithmetic expression, a string expression, a Boolean expression, a set expression, a sequence expression, an entity expression, or a tuple expression:

```
Expression = ArithmExpr | StringExpr | BoolExpr | SetExpr | SeqExpr |
             EntityExpr | TupleExpr
SetExpr = set of Expr
SeqExpr = seq of Expr
```

An arithmetic expression can be an elementary expression, an arithmetic infix expression, or an arithmetic negation:

```
ArithmExpr = Expr | ArithmInfixExpr | ArithmNegation
ArithmInfixExpr :: e1: Expr
                   op: ArithmOp
                   e2: Expr
ArithmOp = { +, -, *, /, ** }
ArithmNegation = ArithmExpr
```

A string expression can be an elementary expression, or string type operation:

```
StringExpr = Expr | StringTypeOp
```

A string type operation is an application of a string type operator to appropriate arguments that returns a string value:

```
StringExpr ::   op: StringOp
                args: seq of Expr
```

A Boolean expression can be a predicate expression, a negation, or a Boolean infix expression:

```
BoolExpr = PredExpr | Negation | BoolInfixExpr
Negation = BoolExpr
BoolInfixExpr ::   b1: BoolExpr
                   op: BoolOp
                   b2: BoolExpr
BoolOp  = { &, | }
```

A predicate expression consists of a pair of arithmetic expressions and a comparison operator:

```
PredExpr :: a1: ArithmExpr
            op: CompOp
            a2: ArithmExpr
CompOp = { =, <, >, <=, >=, != }
```

An entity expression is an elementary expression that returns an entity:

```
EntityExpr = Expr
```

A tuple expression has a tuple type name and a sequence of atomic values:

```
TupleExpr ::    name: TupleTypeName
                values: seq of AtomicValue
```

An elementary expression can be a denotable value, a variable reference, an attribute value, a function invocation, a data model operation that returns a value, or recursively, an "Expression":

```
Expr = DenValue | VarRef | AttribValue | FunInvoc | DataModelOp | Expression
```

The syntactic categories *SetExpr*, *Expr*, *Expression* etc. define the values that can be computed at run-time (dynamically). The set of these values is strictly larger than the set of "denotable values" whose types are defined by *Type* expressions at compile-time (statically). The set of denotable values (as defined in Section 7.3.2. Semantic Domains) includes "atomic values" (basic data type values and entities) and "structured values" (tuples, sets (of atomic values) and sequences (of atomic values)). Since an atomic value can be an entity, arbitrarily complex structures can be built up from a hierarchy of entities. For example, to specify a type "set of set of set of String" define:

```
CLASS S_string = (
 ATTRIBUTE
  comp: {STRING};
)

CLASS Ss_string = (
 ATTRIBUTE
  comp: {S_string};
)

CLASS Sss_string = (
 ATTRIBUTE
  comp: {Ss_string};
)
```

The corresponding denotable values are introduced by the declarations:

```
VAR s: STRING;
  s1: S_String;
  s2: Ss_String;
  s3: Sss_String;
```

An attribute value consists of an attribute name and an entity:

```
AttribValue ::    attrib: AttribName
                  entity: EntityExpr
```

A function invocation consists of the function name, and a sequence of arguments:

```
FunInvoc  :: name: FunName
             args: seq of Expression
```

7.7.2. Semantic Domains

A denotable value can be an atomic value or a structured value:

```
DenValue  =  AtomicValue | StructValue
```

An atomic value can be a basic data type value or an entity:

```
AtomicValue  =  BasicTypeValue | Entity
```

A basic data type value can be an integer, a real, a string, or a Boolean:

```
BasicTypeValue =  Z | R | String | Bool
```

A structured value can be a set value, a sequence value or a tuple value:

StructValue = SetValue | SeqValue | TupleValue
SetValue = set of AtomicValue
SeqValue = seq of AtomicValue

A tuple value is a set of attribute-value pairs:

TupleValue = set of avPair
avPair :: attrib: AttribName
value: AtomicValue

An entity consists of an entity identifier, and a (most specific) class:

Entity :: entID: N_1
class: ClassName

7.7.3. System State

The following domains (abstract types) and variables define the system state. The class aggregation graph is defined in terms of the *HasComp* abstract type:

HasComp :: aggr: ClassName
comp: ClassName

where *aggr* and *comp* fields denote, respectively, the aggregate and component classes. The class generalization hierarchy is defined in terms of the *IsA* abstract type:

IsA :: sub: ClassName
super: ClassName

where *sub* and *super* fields denote, respectively, subclass and superclass. The relation $Is\text{-}a^*$ is a set of *IsA* values. The entity attribute values are defined in terms of an attribute value table entry type:

AvtEntry :: attrib: AttribName
entity: Entity
value: DenValue

where *attrib*, *entity* and *value* fields denote, respectively, attribute name, entity and attribute value. The attribute name is implicitly qualified by the class of the entity. The well-formedness constraint on the attribute name (defined in Section 7.7.4) verifies its correctness subject to the attribute redefinition constraint. The *context conditions environment*

is a mapping from identifiers to the syntactic objects they denote:

$\theta \in$ CCEnv = map Id to { **type, variable, class, procedure, function, attribute, constraint, dataModelOp, entity** }

The *static environment* is a complex object consisting of the set of type, variable, class, procedure and function declarations, and the class generalization and aggregation hierarchies:

$\sigma \in$ SEnv :: types: set of TypeDecl
etValues: map TypeName to SetValue
vars; set of VarDecl
classes: set of ClassDecl
procs: set of ProcDecl
funs: set of FunDecl
isA: set of IsA
hasComp: set of HasComp

The state invariant corresponding to the static environment is:

inv-SEnv(mk-SEnv(types, etValues, vars, classes, procs, funs, isA, hasComp)) Δ
is-DAG(isA)

where the predicate *is-DAG* checks that the directed graph defined by its argument is acyclic.

The *dynamic environment* is a complex object consisting of a store, an attribute value table, a entity class extension map, and an entity database:

$\rho \in$ DEnv :: store: map VarName to [DenValue]
avt: set of AvtEntry
ext: map ClassName to (set of Entity)
entDB: set of Entity

where [DenValue] = DenValue $\cup$ {**null**}. The null denotation represents an undefined value. The state invariant corresponding to the dynamic environment is defined as:

inv-DEnv(mk-DEnv(store, avt, ext, entDB) Δ inv-Avt(avt) & inv-EntDB(entDB)
inv-Avt(avt) Δ $\forall e \in$ avt $\cdot$ is-unique(<attrib(e), entity(e)>)
inv-EntDB(entDB) Δ $\forall e \in$ entDB $\cdot$ is-unique(entID(e)) &
$(\exists C \in$ classes(σ) $\cdot$ $e \in$ ext(C)

The *inv-Avt* predicate checks that there is a unique entry corresponding to each pair of attribute and entity values. The *inv-EntDB* predicate checks that identifiers of entities in

the database are unique, and that they are members of some class extension.

7.7.4. Semantic Well-formedness and Meaning Functions

An enumerated type declaration modifies the static environment by adding to the *types* and *etValues* fields:

```
WF: EnumeratedType → (CCEnv × SEnv) → Bool
WF[mk-EnumeratedType(n, v)](θ, σ) Δ
   θ(n) = type & ¬(mk-EnumeratedType(n, v) ∈ types(σ))

M: EnumeratedType → SEnv → SEnv
M[mk-EnumeratedType(n, v)](σ) Δ
      mk-SEnv((types(σ)∪mk-EnumeratedType(n, v)),
          (etValues(σ)+{n→v}),
          vars(σ), classes(σ), procs(σ), funs(σ), isA(σ), hasComp(σ))
```

A tuple type declaration modifies the static environment by adding to the *types* field:

```
WF: TupleType → (CCEnv × SEnv) → Bool
WF[mk-TupleType(n, as)](θ, σ) Δ
   θ(n) = type & ¬(∃t∈ types(σ) · name(t)=n) & (∀a∈ as) · WFTupleAttrib[a](θ, σ)

M: TupleType → SEnv → SEnv
M[mk-TupleType(n, as)](σ) Δ
      mk-SEnv((types(σ)∪mk-TupleType(n, as)), etValues(σ), vars(σ),
                  classes(σ), procs(σ), funs(σ), isA(σ), hasComp(σ))
```

A variable declaration modifies the the static environment by adding a *VarDecl* object to the *vars* field, and modifies the dynamic environment by adding to the *store* field a null denotation for the variable being declared:

```
WF: VarDecl → (CCEnv×SEnv) → Bool
WF[mk-VarDecl(v, t)](θ, σ) Δ
     θ(v)=variable & θ(t)=type} & ¬(mk-VarDecl(v, t)∈ vars(σ))

M: VarDecl → (SEnv×DEnv) → (SEnv×DEnv)
M[mk-VarDecl(v, t)](σ, ρ) Δ
      let σ' = mk-SEnv(types(σ), etValues(σ), (vars(σ)∪mk-VarDecl(v, t)),
                  classes(σ), procs(σ), funs(σ), isA(σ), hasComp(σ)) in
       let ρ' = mk-DEnv((store(ρ)+{v→null}), avt(ρ), ext(ρ), entDB(ρ)) in
      (σ', ρ')
```

A class declaration modifies the static environment by updating its *classes*, *isA* and *hasComp* fields:

WF: ClassDecl → (CCEnv×SEnv) → Bool
WF[mk-ClassDecl(n, as, cs, super, sub)](θ, σ) Δ
θ(n)=**class** & ¬(∃C∈ classes(σ) · name(C)=n) &
∀a∈ as · WFAttrib[a](n, θ, σ) & ∀c∈ cs · WFConstr[c](θ) &
∀p∈ super · WFIsA[mk-IsA(n, p)](θ) & ∀b∈ sub · WFIsA[mk-IsA(b, n)](θ)

M: ClassDecl → SEnv → SEnv
M[mk-ClassDecl(n, as, cs, super, sub)](σ) Δ
mk-SEnv(types(σ), etValues(σ), vars(σ), (classes(σ)∪
{mk-ClassDecl(n, as, cs, super, sub)}, procs(σ), funs(σ),
isA(σ)∪ {mk-IsA(n, *n'*) | *n'*∈ super }∪ {mk-IsA(*n'*, n) | *n'*∈ sub}),
(hasComp(σ)∪
{mk-HasComp(n, *n'*) | ∃a∈ as · dom(a)∈ name(classes(σ))}))

The well-formedness predicate for the *IsA* object ensures that both the arguments are class names (i.e. **class** (syntactic) objects).

WF: IsA → CCEnv → Bool
WF[mk-IsA(n_1, n_2)](θ) Δ θ(n_1) = θ(n_2) = **class**

The well-formedness constraint on attribute definitions says that the attribute name is either not inherited from any superclass, or if inherited, satisfies the attribute redefinition constraint (Section 4.).

WF: Attrib → (ClassName×CCEnv×SEnv) → Bool
WF[mk-Attrib(n, d)](C, θ, σ) Δ θ(n)=**attribute** & θ(d)=**type** &
¬(n∈ inh_attrs(C, σ)) |
∀a∈ {a_{inh} | ∃C_1 · a_{inh} ∈ attrib(C_1) & C_1 ∈ classes(σ) &
name(C_1) ∈ super(C) } · d ⊆ dom(a)

where the function *inh_attrs* (inherited attributes) is defined as:

inh_attrs(n, σ) Δ
for C∈ classes(σ) such that name(C)=n ·
if (super(C)= Φ)
then Φ
else {name(a) | ∃C_1∈ classes(σ) · a∈ attrib(C_1) & name(C_1)∈ super(C)} ∪
inh_attrs(name(C_1), σ)

A procedure declaration modifies the static environment by adding to the set of procedure declarations, and modifies the dynamic environment by updating the *store* to include a (recursively) defined function obtained by computing the denotations of the statements in the procedure with the input parameters bound to values in the calling environment, and

the output parameter variables and the local variables bound to null denotations in the *store*.

```
WF: ProcDecl → (CCEnv×SEnv) → Bool
WF[mk-ProcDecl(n, ips, ops, lts, lvs, b)](θ, σ) Δ
      all-disjoint(<elems(ips), elems(ops), elems(lvs), dom(θ)>) &
      length(ips)≥0 & length(ops)≥0 & length(lvs)≥0 &
      ¬(∃p∈ procs(σ) · name(p)=n)) &
      ∀t∈ lts · WFTypeDecl[t](θ, σ) &
      ∀v∈ lvs · WFVarDecl[v](θ, σ) &
      WFBody[b](θ, σ)

M: ProcDecl → (SEnv×Denv) → (SEnv×(DEnv→DEnv))
M[mk-ProcDecl(n, ips, ops, lts, lvs, b)](σ, ρ) Δ
      let σ' = mk-SEnv(lts∪types(σ), etvalues(σ),
                       (vars(σ)∪ips∪ops∪lvs), classes(σ),
                       (procs(σ)∪mk-ProcDecl(n, ips, ops, lts, lvs, b)),
                       funs(σ), isA(σ), hasComp(σ)) in
       (let f(store(ρ')) =
        (let ρ' = mk-DEnv(store(ρ)+{id →MExpression[id](ρ) | id ∈ ips} +
                          {id→null | id∈ ops∪lvs}, avt(ρ), ext(ρ), entDB(ρ)) in
         for i=1 to length(b) do MStmnt[b[i]](σ', ρ');
        ) in
        f
       ) in
      (σ', f)
```

where *length* is the sequence operator that returns the length of a sequence; *elems* is the sequence operator that generates the set of elements in the sequence; and the predicate *all-disjoint* is defined as:

```
all-disjoint: seq of (set of Id) → Bool
all_disjoint(ss) Δ ∀i,j∈ indxs(ss) · i≠j => ss(i) ∩ ss(j) = Φ
```

where *indxs* is the sequence operator that returns the set of index values of the sequence elements.

A function declaration modifies the static environment by adding to the set of function declarations, and modifies the dynamic environment by updating the *store* to include a (recursively) defined function obtained by computing the denotations of the statements in the functions with the input parameters bound to values in the calling environment, and the local variables bound to null denotations in the *store*.

```
WF: FunDecl → (CCEnv × SEnv) → Bool
WF[mk-FunDecl(n, ps, rt, lts, lvs,  b)](θ, σ) Δ
   all-disjoint(<elems(ps), elems(lvs), dom(θ)>) &
   length(ps)≥0 & length(lvs)≥0 &
   ¬(∃f∈ funs(σ) · name(f)=n) &
   ∀t∈ lts · WFTypeDecl[t](θ, σ) &
   ∀v∈ lvs · WFVarDecl[v](θ, σ) &
   WFBody[b](θ, σ)

M: FunDecl → (SEnv×Denv) → (SEnv×(DEnv→DEnv))
M[mk-FunDecl(n, ps, rt, lts, lvs, b)](σ, ρ) Δ
      let σ' = mk-SEnv((types(σ)∪lts, vars(σ)∪ps∪lvs), classes(σ),
           procs(σ), (funs(σ)∪mk-FunDecl(n, ps, rt, lts, lvs,  b)),
           isA(σ), hasComp(σ)) in
       (let f(store(ρ')) =
        (let ρ' = mk-DEnv(store(ρ)+{id →MExpression[id](ρ) | id∈ ps} +
                          {id→null | id∈ lvs}, avt(ρ), ext(ρ), entDB(ρ) in
         for i=1 to length(b) do MStmnt[b[i]](σ', ρ');
        ) in
        f
       ) in
      (σ', f)
```

An assignment modifies the dynamic environment by updating the *store* with a new element that maps the variable (on the left hand side of the assignment) to the value obtained by computing the denotation of the expression (on the right hand side of the assignment):

```
WF: Assgn → (CCEnv×SEnv) → Bool
WF[mk-Assgn(v, e)](θ, σ) Δ θ(v) = variable  &  WFExpression[e](θ, σ)

M: Assgn → DEnv → DEnv
M[mk-Assgn(v, e)](ρ) Δ
    mk-DEnv((store(ρ)+{v→MExpression[e](ρ)}), avt(ρ), ext(ρ), entDB(ρ))
```

A procedure invocation modifies the dynamic environment by applying the function obtained by computing the denotation of the procedure to the updated store that includes the denotations of the arguments:

```
WF: ProcInvoc → (CCEnv×SEnv) → Bool
WF[mk-ProcInvoc(n, in-args, out-args)](θ, σ) Δ
    θ(n) = procedure & ∀e∈ elems(in-args) · WFExpression[e](θ, σ) &
    ∀v∈ elems(out-args) · WFVarDecl[v](θ, σ)

M: ProcInvoc → DEnv → DEnv
M[mk-ProcInvoc(n, in-args, out-args)](ρ) Δ
  if (for i=1 to length(in-args), o=ips[i], e=in-args[i] · DefState(o) ⊆ DefState(e))
    then
      let P = MProcDecl[n](σ, ρ) in
        let ρ'= (mk-DEnv(store(ρ)+<ips[i]→MExpression[in-args[i]](ρ)|
                          i∈ indxs(in-args)>+ <out-args[i]→null | i∈ indxs(out-args)>),
                        avt(ρ), ext(ρ), entDB(ρ))
          in P(σ, ρ')
    else
      ρ
```

where

DefState(o) = {A | ∃C∈ classes(σ) · name(C)=type(o) &
a∈ attrib(C) & name(a)=A & value of attribute A is required to be non-null}

DefState(e) = {A | ∃C∈ classes(σ) · name(C)=class(e) &
a∈ attrib(C) & name(a)=A & ¬(A(e)=null)}

A function invocation returns a value obtained by applying the function obtained by computing the denotation of the function to the updated store that includes the denotations of the arguments:

```
WF: FunInvoc → (CCEnv×SEnv) → Bool
WF[mk-FunInvoc(n, args)](θ, σ) Δ
    θ(n) = function & ∀e∈ elems(args) · WFExpression[e](θ, σ)

M: FuncInvoc → DEnv → DenValue
M[mk-FunInvoc(n, args)](ρ) Δ
  if (for i=1 to length(args), p=ps[i], e=args[i] · DefState(p) ⊆ DefState(e))
    then
      let F = MFunDecl[n](ρ) in
        let ρ'= (mk-DEnv(store(ρ)+<ps[i]→MExpression[args[i]](ρ)|
                          i∈ indxs(in-args)>, avt(ρ), ext(ρ), entDB(ρ))
          in F(σ)(ρ')(args)
    else
      null
```

where

DefState(o) = {A | ∃C∈ classes(σ) · name(C)=type(o) &
a∈ attrib(C) & name(a)=A & value of attribute A is required to be non-null}

DefState(e) = {A | ∃C∈ classes(σ) · name(C)=class(e) &
a∈ attrib(C) & name(a)=A & ¬(A(e)=**null**)}

A for-do iteration loop modifies the dynamic environment by updating its store by adding an element that maps the loop variable to the denotation generated by the iterator computing the denotations of the statement in the loop in the updated dynamic environment:

WF: ForDo → (CCEnv×SEnv) → Bool
WF[mk-ForDo(v, it, b)](θ, σ) Δ
θ(v) = **variable** & WFIterator[it](θ, σ) & ∀s∈ b · WFStmnt[s](θ, σ)

M: ForDo → DEnv → DEnv
M[mk-ForDo(v, it, b)](ρ) Δ
for ∀i∈ MIterator[it](ρ) do
MStmnt[b](mk-DEnv((store(ρ)+{v→i}), avt(ρ), ext(ρ), entDB(ρ));

An if-then-else construct computes the denotation of the Boolean expression in its condition part; if it evaluates to TRUE, the denotation of the statement sequence in the then part is computed, otherwise the denotation of the statement sequence in the else part is computed:

WF: IfThenElse → (CCEnv×SEnv) → Bool
WF[mk-IfThenElse(cond, then_part, else_part)](θ, σ) Δ
WFBoolExpr[cond](θ, σ) & ∀s∈ elems(then_part) · WFStmnt[s](θ, σ) &
∀s∈ (elems(else_part) · WFStmnt[s](θ, σ)

M: IfThenElse → DEnv → DEnv
M[mk-IfThenElse(cond, then_part, else_part)](ρ) Δ
if MBoolExpr[cond](ρ) **then** MStmnt[then_part](ρ) **else** MStmnt[else_part](ρ)

The semantics of an if-then construct are similarly defined as:

WF: IfThen → (CCEnv×SEnv) → Bool
WF[mk-IfThen(cond, then_part](θ, σ) Δ
WFBoolExpr[cond](θ, σ) & ∀s∈ elems(then_part) · WFStmnt[s](θ, σ)

M: IfThen → DEnv → DEnv
M[mk-IfThen(cond, then_part)](ρ) Δ
if MBoolExpr[cond](ρ) **then** MStmnt[then_part](ρ) **else** ρ

A while-do loop computes the application of a recursive function (wh) obtained by

evaluating the denotation of the Boolean expression in its condition part and testing if it is TRUE:

WF: WhileDo → (CCEnv×SEnv) → Bool
WF[mk-WhileDo(cond, body)](θ, σ) Δ
    WFBoolExpr[cond](θ, σ) & ∀s∈ body · WFStmnt[s](θ, σ)

M: WhileDo → DEnv → DEnv
M[mk-WhileDo(cond, body)](ρ) Δ
    let wh = (def bv: MBoolExpr[b]; if bv then (MBoolExpr[b]; wh) else I_{DEnv}) in
    wh(ρ)

A variable reference returns the value to which it is mapped by the store:

WF: VarRef → (CCEnv×SEnv) → Bool
WF[mk-VarRef(v)](θ, σ) Δ θ(v)=**variable** & (v∈ vars(σ))

M: VarRef → DEnv → DenValue
M[mk-VarRef(v)](ρ) Δ store(ρ)(v)

An attribute value is obtained by retrieving the value component of the attribute value table entry such that its attribute and entity components correspond to the given attribute and the entity obtained by computing the given entity expression:

WF: AttribValue → (CCEnv×SEnv) → Bool
WF[mk-AttribValue(n, e)](θ, σ) Δ θ(n)=**attribute** & WFEntityExpr[e](θ, σ)

M: AttribValue → DEnv → DenValue
M[mk-AttribValue(n, e)](ρ) Δ
    value(k) such that ∃k∈ avt(ρ) · attrib(k)=n & entity(k)=MEntityExpr[e](ρ)

The semantics of an arithmetic infix expression are defined as:

```
WF: ArithmInfixExpr → (CCEnv×SEnv) → Bool
WF[mk-ArithmInfixExpr(e1, op, e2)](θ, σ) Δ
    WFArithmExpr[e1](θ, σ) & op ∈ ArithmOP & WFArithmExpr[e2](θ, σ)

M: ArithmInfixExpr → DEnv → AtomicValue
M[mk-ArithmInfixExpr(e1, op, e2)](ρ) Δ
      let v1 = MArithmExpr[e1](ρ) in
        let v2 = MArithmExpr[e2](ρ) in
       cases op:
              + → v1 + v2
              - → v1 - v2
              * → v1 * v2
              / → v1 / v2
              ** → v1 ** v2
```

The semantics of a arithmetic negation are defined as:

```
WF: ArithmNegation → (CCEnv×SEnv) → Bool
WF[mk-ArithmNegation(e)](θ, σ) Δ WFArithmExpr[e](θ, σ)

M: ArithmNegation → DEnv → AtomicValue
M[mk-ArithmNegation(e)](ρ) Δ - MArithmExpr[e](ρ)
```

The semantics of a Boolean infix expression are defined as:

```
WF: BoolInfixExpr → (CCEnv×SEnv) → Bool
WF[mk-BoolInfixExpr(e1, op, e2)](θ, σ) Δ
      WFBoolExpr[e1](θ, σ) & op ∈ BoolOP & WFBoolExpr[e2](θ, σ)

M: BoolInfixExpr → DEnv → Bool
M[mk-BoolInfixExpr(e1, op, e2)](ρ) Δ
     let v1 = MBoolExpr[e1](ρ) in
       let v2 = MBoolExpr[e2](ρ) in
         cases op:
               AND → if v1 then v2 else false
               OR  → if v1 then true else v2
```

The semantics of a Boolean negation are defined as:

```
WF: Negation → (CCEnv×SEnv) → Bool
WF[mk-Negation(e)](θ, σ) Δ  WFBoolExpr[e](θ, σ)

M: Negation → DEnv → Bool
M[mk-Negation(e)](ρ) Δ if MBoolExpr[e](ρ) then false else true
```

The semantics of a predicate expression are defined as:

```
WF: PredExpr → (CCEnv×SEnv) → Bool
WF[mk-PredExpr(a1, op, a2)](θ, σ) Δ
   WFArithmExpr[a1](θ, σ) & op ∈ CompOp & WFArithmExpr[a2](θ, σ)

M: PredExpr → DEnv → Bool
M[mk-PredExpr(a1, op, a2)](ρ) Δ
   let v1 = MArithmExpr[a1](ρ) in
     let v2 = MArithmExpr[a2](ρ) in
   cases op:
         = → if v1=v2  then true else false
         > → if v1>v2  then true else false
         < → if v1<v2  then true else false
         >= → if v1≥v2  then true else false
         <= → if v1≤v2  then true else false
         != → if v1≠v2  then true else false
```

7.8. Summary

This chapter has discussed formal specification techniques. VDM is a model-based (denotational) specification technique that has been used successfully in a number of actual software design and development projects. Larch is an algebraic (axiomatic) specification technique supporting software design and formal verification using a hierarchy of data and procedural abstractions. Both VDM and Larch specifications of a structured type system are presented. An implementation of the values and operators of the structured type system (the SVIM subsystem) is also presented. A sketch of the formal specification of the semantic data model and the integrated database language (IDL) is then presented using denotational semantics techniques.

Chapter 8

CONCLUSION

8.1. Summary

This thesis has focused on the problem of applying database technology to computer-aided design applications. Chapter 1 provided a brief overview of computer-aided design systems and integrated design environments, and the motivation for design databases. Chapter 2 elaborated on the special database requirements of CAD applications and the limitations of conventional database systems in meeting these requirements. It also proposed a design database system architecture consisting of: a conceptual-level, semantic data model; an integrated database language; complex application-oriented operations; complex semantic integrity and consistency constraints; and a concurrent access control and interactive design transaction management subsystem (not carried out in detail). Chapter 3 surveyed related research efforts on some of these topics. Chapter 4 described the design of the conceptual-level, semantic data model which captures the structural and behavioural abstractions of design databases in terms of entities, entity classes, entity interrelationships, integrity constraints and operations. Chapter 5 discussed database language design issues of data abstraction, polymorphism, data persistence, type system implementation, and system architecture in the context of several current research proposals. This led to the conceptual basis for the design of an integrated database language followed by a specification of the features of the language. Chapter 6 presented an illustrative example of the specifications for defining and processing the design of a VLSI 4-bit adder circuit. Chapter 7 dealt with formal specification issues. Model-based (operational), algebraic (axiomatic) and denotational semantics specification techniques were discussed. The structured type system that would subsequently be used in the specification of the semantic data model and the integrated database language was then

formally defined in terms of VDM and Larch specifications. Using these structured types, a denotational semantics specification for the model and the language was then outlined in terms of an abstract syntax (syntactic domains), a collection of semantic domains, and a collection of well-formedness (context conditions), and meaning (semantics) functions.

8.2. Future Research

In the context of the work reported in this thesis, several additional aspects and related issues of design database systems can be identified for future research. Some of these are discussed in the following subsections.

8.2.1. DBMS/Design Tool Interface

The user interface provided by the integrated database language is based directly on the underlying semantic data model. Thus, at the level of this interface, the user deals directly with the abstractions of entities, entity classes, structured values etc. A design tool written in the integrated database language can therefore directly access and manipulate these abstractions. However, there is a large collection of already existing design tools written in other languages. To be able to use these tools with the design database an interface between the design tool and the DBMS has to be defined and implemented. This interface translates between the semantic data model abstractions of the DBMS and the data structuring, manipulation and transmission abstraction provided by the design tool (application) language. A separate interface will be required for each such language. This interface can take either of the two forms: (1) a set of design tool callable DBMS routines that can be linked to the design tool before execution; or, (2) an independently executing *DBMS Interface* process that mediates communication between the design tool and the DBMS by appropriately translating between the two models. Designing and implementing such interfaces for the design database system described in this thesis constitutes a

topic for future work.

8.2.2. Database System Services

The additional system services needed in practical database systems include: concurrent access control and design transaction management; multiple version management; performance optimization; reliability and failure recovery; and distributed processing architectures. Some of these topics are considered below:

Concurrent Access Control and Transaction Management

In a design database, the granularity of concurrent access is the entity rather than a physical (storage block or file) or a logical (record or tuple) unit [BKK85, Kat86]. Since the entities are shared, structured hierarchically and interrelated in complex ways, the semantics of concurrent access are much more complicated than in conventional databases. Consequently the classical concurrent access control (locking and time stamping), transaction atomicity (commit/rollback), and failure recovery (undo/redo) protocols are inadequate. A *transaction* is a sequence of data manipulation and update operations that maintains the integrity and consistency of the database. A transaction is atomic (i.e. all operations in the sequence are executed to completion or none of the operations is executed), and durable (i.e. system failures are transparent with respect to the transaction). Several recent papers have reported on concurrent access and transaction management models: Katz [Kat86] discusses a model based on the check-out/check-in "library operations". Bancilhon et al [BKK85] propose a model of design database transaction management based on a hierarchy of cooperating client/subcontractor transactions. These can be adapted and extended to work with database model presented in this thesis.

Version Management

Version management is another important service that will greatly aid the designer

by automating much of the tedious accounting information necessary to manage multiple versions. Zdonik [ZdW86] discusses a version control mechanism that captures the "derived-from" relationship between versions. Beech and Mahbod [BeM88] describe a version creation and referencing model for the Iris [Fis87] object-oriented database system. Katz and Lehman [KaL84] discuss secondary storage structures to support and implement versions and alternatives of large design files. Again, these models can be adapted in an implementation of the version-of relationship provided by the semantic data model specified.

Performance Optimization

Performance is another issue that is crucial to effective design databases. Without adequate speed and fast response to user-invoked operations and queries a design DBMS will fail to meet the performance requirements of an interactive design system. Database queries which involve iterating over recursively defined objects arise often in design databases; for example, in querying recursively defined part hierarchies [RHM], and in checking recursively defined semantic integrity constraints (like *BoxOverlap*). Such queries can be computationally very expensive unless some sophisticated optimization techniques are employed that significantly reduce the search space. A large class of such queries involves computing the "transitive closure" of a relation, or the least fixpoint of a recursively defined equation. Several such optimization techniques have been investigated in recent literature and these can also be adapted in the context of the model presented in this thesis.

8.2.3. Other Data Models

NF^2 Relational Models

Non-first normal form (NF^2) relational models [AMM82, JaS82] relax the first normal form constraint of the classical relational model requiring attribute values to be atomic by allowing them to be relations. A further generalization discussed in [PiA86] allows attribute values to be tuples, lists or multisets so that arbitrarily complex structures can be built up by suitably nesting these basic constructors. A NF^2 algebra which defines nest and unnest operations, assignment and ordering of values provides the basic operations of the data model. An extended SQL-like query, data definition and data manipulation language provides a user interface to a NF^2 database. [Dad86] describes a prototype DBMS based on the NF^2 model. The use of the NF^2 model to provide an intermediate stage in translating between the semantic data model view of the design DBMS and a relational view of a user or design tool, or in data sharing and exchange in a "federated database system" [Zeh86] appears feasible, and provides a topic for future research.

Knowledge-base Systems

The quest for making database systems more "intelligent" by using AI techniques has led to the concepts of knowledge representation paradigms and knowledge-base systems. The knowledge representation models include semantic nets and frame-based systems which incorporate a rich set of (possibly user-defined) object interrelationships and property inheritance hierarchies [Tic87]. Current research focuses on integrating these concepts with the object-oriented programming paradigm (object classes, methods and messages) to yield the so called knowledge-base systems that combine the reasoning ability of AI systems with the efficient data structuring and algorithmic processing ability of database systems.

Logic Databases

There has been much research exploring the connection between first-order logic and the relational data model. A logic database views a relation as an extensionally defined predicate. Tuples in the relation correspond to elementary facts, which are predicates with constant arguments (corresponding to the tuple components) that evaluate to TRUE. A query corresponds to an open wff (i.e. a well-formed formula with free variables). Database retrieval corresponds to proving the query goal by binding the free variables to values in the database. A large class of structural integrity constraints defined by functional and multivalued dependencies can be specified as closed Horn clauses with no function symbols. In such a system, recursively defined objects and queries can be formulated using Horn clauses. The advantages of the logic data model include: a unified underlying formalism for data definition, integrity constraint specification, data manipulation and querying; and, a more expressive and powerful computational model that permits recursion. Some of the limitations that need to be addressed by further investigations include: more efficient secondary storage management; support for concurrency, failure recovery and security; data typing, polymorphism and inheritance; more efficient evaluation strategies; and optimization strategies for query processing.

Zaniolo [Zan85] proposes an extension of the relational algebra that deals with complex objects defined as nested predicates (i.e. predicates whose arguments may themselves be predicates) and provides deductive retrieval using non-recursive, safe Horn clauses. Bancilhon [Ban86] proposes an integration of the object paradigm with the logic paradigm to provide a model that can deal with "complex (structured) objects" and incremental update (or modification) operations. A recent paper by Ait-Kaci [Ait86] extends the first-order logic programming paradigm by introducing the notion of types and "type subsumption" which promises fuller integration of object-oriented databases (abstract

types, polymorphism and inheritance) with logic programming (deductive processing by unification). Tsur and Zaniolo [TsZ86] propose a logic database language called LDL that augments PROLOG by providing pure Horn clause logic (with sequential rule execution constraint removed), sets as atomic objects, negation by set difference, schema definition and update capabilities.

The logic model has applications to several database issues including deductive processing and inferencing, handling incomplete information, and database programming languages. Investigation of these issues in the context of the design database model presented in this thesis is another direction for future research.

References

[Abr74] J.R. ABRIAL, Data Semantics , in *Data Base Management*, J.W. Klimbi and K.L. Koffeman (ed.), North Holland, 1974.

[Abr80] J.R. ABRIAL, The Specification Language Z: Basic Library, Working Paper, Oxford University Programming Research Group, 1980.

[Ait86] H. AIT-KACI, Type Subsumption as a Model of Computation in Expert Database Systems, in *Expert Database Systems*, L. Kerschberg (ed.), 1986.

[Alb84] A. ALBANO, Type Hierarchies and Semantic Data Models, *SIGPLAN Notices Notices*, 1984.

[ACO85] A. ALBANO, L. CARDELLI and R. ORSINI, Galileo: A Strongly-Typed Interactive Conceptual Language, *ACM Trans. Database Systems 10*, 2 (June 1985), 230-260.

[AMM82] H. ARISAWA, K. MORIYA and T. MIURA, Operations and Properties of Non-first-normal-form Relational Databases, *Principles of Database Systems Symposium Proceedings*, 1982.

[AGS84] W.W. ARMSTRONG, M. GREEN and P. SRIRANGAPATNA, A Database Management System and Associated Tools for a General Design Environment, *Proc of 1984 Canadian Conf on VLSI*, Oct 1984.

[ABC83] M.P. ATKINSON, P.T. BAILEY and K.J. CHISOLM, An Approach to Persistent Programming, *Computer Journal 26*, 4 (November 1983), .

[BaT80] C.M. BAKER and C.J. TERMAN, Tools for Verifying Integrated Circuit Designs, *Lambda 1*, 3 (4th Quarter 1980), 22-30.

[BKK85] F. BANCILHON, W. KIM and H.F. KORTH, A Model of CAD Transactions, *Proceedings of VLDB* , 1985.

[Ban86] F. BANCILHON, A Logic-programming/Object-oriented Cocktail, *SIGMOD Record 15*, 3 (September 1986), .

[Bar84] D.R. BARSTOW, *Interactive Programming Environments*, McGraw-Hill Inc., 1984.

[BaK85] D.S. BATORY and W. KIM, Modeling Concepts for VLSI CAD Objects, *Supplement to 1985 ACM SIGMOD Proceedings*, 1985, 18-32.

[BeM88] D. BEECH and B. MAHBOD, Generalized Version Control in an Object-oriented Database, *OOPSLA Proceedings*, 1988.

[BeN71] C.G. BELL and A. NEWELL, *Computer Structures: Readings and Examples*, McGraw Hill, New York, 1971.

[BjJ82] D. BJOERNER and C.B. JONES, *Formal Specification and Software Development*, Prentice Hall, 1982.

[Bro80] M.L. BRODIE, The Application of Data Types to Database Semantic Integrity, *Information Systems 5*, (1980), 287-296.

[Bro81] M.L. BRODIE, Data Abstraction for Designing Database-Intensive Applications, *SIGMOD Record 11*, 2 (Feb 1981), .

[BuN84] P. BUNEMAN and R. NIKHIL, The Functional Data Model and its Uses for Interaction with Databases, in *On Conceptual Modeling*, M.L. Brodie et.al (ed.), Springer Verlag, 1984.

[BuA86] P. BUNEMAN and M. ATKINSON, Inheritance and Persistence in Database Programming Languages, *Proceedings of SIGMOD 86*, May 1986.

[CCC81] *CHILL Formal Definition*, CCITT Period 1980-1984 Working Party XI/3, 1981.

[Car84] L. CARDELLI, Amber, Technical Report, AT&T Bell Labs, 1984.

[CaW85] L. CARDELLI and P. WEGNER, On Understanding Types, Data Abstraction and Polymorphism, *Computing Surveys 17*, 4 (December 1985), .

[Cha76] D.D. CHAMBERLIN, SEQUEL2: A Unified Approach to Data Definition, Manipulation and Control, *IBM Journal of Research and Development 20*, 6 (1976), 560-575.

[Che76] P.P.S. CHEN, The Entity-Relationship Model: Toward a Unified view of Data, *ACM Trans. Database Systems 1*, 1 (March 1976), 9-36.

[Chu83] K.C. CHU, vdd - A VLSI Design Database System, *Engineering Design Applications: Proc. of Annual Meeting, Database Week*, May 1983.

[Cle81] E. CLEMONS, Design of an External Schema Facility to Define and Process Recursive Structures, *ACM Transactions on Database Systems 6*, 2 (June 1981), 295-311.

[Cod79] E.F. CODD, Extending the Database Relational Model, *ACM Trans. Database Systems 4*, 4 (December 1979), 397-434.

[Cod81] E. F. CODD, Data Models in Database Management, *SIGMOD Record 11*, 2 (Feb 1981), .

[CHJ86] B. COHEN, W.T. HARWOOD and M.I. JACKSON, *The Specification of Complex Systems*, Addison-Wesley Publihing Company, 1986.

[CoM84] G. COPELAND and D. MAIER, Making Smalltalk a Database System, *PODS Symposium Proceedings*, 1984, 316-325.

[Dad86] P. DADAM, A DBMS to Support Extended NF2 Relations: An Integrated View on Flat Tables and Hierarchies, *Proceedings of SIGMOD 1986*, May 1986.

[DMN70] O.J. DAHL, B. MYRHAUG and K. NYGAARD, SIMULA Common Base Language, S-22, Norwegian Computing Centre, 1970.

[Dep81] DEPARTMENT OF INDUSTRY(UK), *Report on the Study of Ada-based System Development Methodology*, Department of Industry, 1981.

[DKL85] N. DERRETT, W. KENT and P. LYNGBAEK, Some Aspects of Operations in an Object-oriented Database, *Database Engineering 8*, 4 (December 1985), 66-74, IEEE TC on Database Engineering.

[DiL85] K.R. DITTRICH and R.A. LORIE, Object-oriented Database Concepts for Engineering Applications, RJ 4691, IBM Research Laboratory, San Jose, May 1985.

[EaT79] C. EASTMAN and R. THORNTON, *A Report on GLIDE2 Language Definition*, CAD Group, Institute of Physical Planning, Carnegie Mellon University, March 1979.

[Fis87] D.H. FISHMAN, Iris: An Object-Oriented Database Management System, *ACM Transactions on Office Information Systems 5*, 1 (January 1987), 48-69.

[Gib84] S.J. GIBBS, An Object-oriented Office Data Model, CSRG-154, Computer Systems Research Group, University of Toronto, January 1984.

[GoR83] A. GOLDBERG and D. ROBSON, *SMALLTALK-80: The Language and its Implementation*, Addison-Wesley, 1983.

[Gor79] M.J.C. GORDON, *The Denotational Description of Programming Languages: An Introduction*, Springer, New York, 1979.

[Gri76] D. GRIES, An Illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs, *IEEE Transactions on Software Engineering 2*, (December 1976), 238-244.

[Gri81] D. GRIES, *The Science of Programming*, Springer Verlag, 1981.

[Gua78] L.R. GUARINO, The Evolution of Abstraction in Programming Languages, CMU-CS-78-128, Department of Computer Science, Carnegie Mellon University, 22 May 1978.

[Gut80] J.V. GUTTAG, Notes on Type Abstraction, *IEEE Transactions on Software Engineering SE-6*, 1 (January 1980), 13-23.

[GHW85] J.V. GUTTAG, J.J. HORNING and J.M. WING, Larch in Five Easy Pieces, Report #5, Digital Systems Research Center, Palo Alto, CA, July 1985.

[HHB85] HHB, *CADAT User's Manual Revision 5.0*, HHB Softron, Mahwah, NJ, June 1985.

[HaL84] G. HALLMARK and R.A. LORIE, Towards VLSI Design Systems using Relational Databases, *Proceedings of Compcon 84*, 1984, 326-329.

[HaM81] M. HAMMER and D. MCLEOD, Database Description with SDM: A Semantic Database Model, *ACM Tods 6*, 3 (September 1981), 351-386.

[Har84] M. HARDWICK, Extending the Relational Database Data Model for Design Applications, *ACM/IEEE 21st Design Automation Conference Proceedings*, 1984, 110-116.

[JaS82] G. JAESCHKE and H.J. SCHEK, Remarks on the Algebra of Non First Normal Form Relations, *Proceedings of SIGMOD PODS* , March 1982.

[Joh80] H.R. JOHNSON, The Engineering Data Management System for IPAD, in *IPAD: Integrated Programs for Aerospace Vehicle Design*, 1980, 145-178.

[Jon86] C.B. JONES, *Systematic Software Development Using VDM*, Prentice Hall, 1986.

[Kat82] R.H. KATZ, A Database approach to managing VLSI Design Data, *ACM/IEEE 20th Design Automation Conf Proc*, 1982, 274-282.

[KaW83] R.H. KATZ and S. WEISS, Chip Assemblers: Concepts and Capabilities, *20th Design Automation Conference Proceedings*, 1983, 25-30.

[KSS83] R. KATZ, W. SCACCHI and P. SUBRAHMANYAM, Development Environments for VLSI and Software Engineering, *VLSI and Software Engineering Workshop*, 1983, 50-63.

[KaL84] R.H. KATZ and T.J. LEHMAN, Database Support for Versions and Alternatives of Large Design Files, *IEEE Transactions on Software Engineering SE-10*, 2 (March 1984), 191-200.

[Kat86] R.H. KATZ, Computer-Aided Design Databases, in *New Directions for Database Systems*, G. Ariav & J. Clifford (ed.), Ablex Publishing Corporation, 1986.

[Kel84] K.H. KELLER, An Electronic Circuit CAD Framework, Ph.D. Thesis, Department of Electrical Engineering and Computer Science, UC Berkeley, 1984.

[Ker81] M. L. KERSTEN, The Architecture of the PLAIN Database Handler, *Software Practice and Experience 11*, (1981), 175-186.

[Kla85] P. KLAHOLD, A Transaction Model Supporting Complex Applications in Integrated Information Systems, *Proceedings of ACM 1985 International Conference on Management of Data*, May 1985, 388-401.

[KuA86] K.G. KULKARNI and M.P. ATKINSON, EFDM: Extended Functional Data Model, *The Computer Journal 29*, 1 (1986), 38-46.

[LeO83] C.L. LEATH and S.J. OLLANIK, Software Architecture for the Implementation of a Computer Aided Engineering System, *ACM/IEEE 20th Design Automation Conference Proceedings*, 1983, 137-142.

[LiG86] B. LISKOV and J. GUTTAG, *Abstraction and Specification in Program Development*, The MIT Press, McGraw-Hill Book Company, 1986.

[Lon78] R. LONDON ET AL, Proof Rules for the Programming Language EUCLID, *Acta Inf. Informatica 7*, (1978), .

[LoP83] R.A. LORIE and W. PLOUFFE, Complex Objects and their Use in Design Transactions, *Engineering Design Applications: Proceedings of ACM/IEEE Database Week*, May 1983.

[MSR84] M. MALL, J.W. SCHMIDT and M. REIMER, Data Selection, Sharing, and Access Control in a Relational Scenario, in *On Conceptual Modelling*, M.L. Brodie, J. Mylopoulos & J.W. Schmidt (ed.), Springer-Verlag, 1984, 411-440.

[Mar82] J. MARTIN, *Program Design which is Provably Correct*, Savant Institute, Carnforth, Lancashire, UK, November 1982.

[Mat85] D.C.J. MATTHEWS, Poly and Standard ML, *ACM SIGPLAN Notices Notices 20*, 9 (September 1985), .

[McG80] A.D. MCGETTRICK, *The Definition of Programming Languages*, Cambridge University Press, 1980.

[McG86] N. Gehani & A.D. McGettrick, ed., *Software Specification Techniques*, Addison Wesley, 1986.

[McS81] D. MCLEOD and J. M. SMITH, Abstraction in Databases, *SIGMOD Record 11*, 2 (Feb 1981), .

[McW78] T. MCWILLIAMS and L. WIDDOES JR., SCALD: Structured Computer-Aided Logic Design, Technical Report No. 152, Digital Systems Laboratory, Stanford University, March 1978.

[MeC80] C. MEAD and L. CONWAY, *Introduction to VLSI Systems*, Addison-Wesley, Reading, 1980.

[Mer78] T.H. MERRETT, Aldat - Augmenting the Relational Algebra for Progammers, SOCS-78-1, School of Computer Science, McGill University, 1978.

[Mil84] R. MILNER, A Proposal for Standard ML, *Proceedings of the Symposium on Lisp and Functional Programming*, 1984, 184-197.

[MBW80] J. MYLOPOULOS, P.A. BERNSTEIN and H.K.T. WONG, A Language Facility for Designing Database-Intensive Applications, *ACM Transactions on Database Systems 5*, 2 (June 1980), 185-207.

[Nag75] L.W. NAGEL, SPICE2: A Computer Program to Simulate Semiconductor Circuits", ERL-M520, University of California, Berkeley, May 1975.

[New81] A. NEWTON, Design Aids for VLSI: The Berkeley Perspective, *IEEE Transactions on Circuits and Systems CAS-28*, 7 (July 1981), .

[O'79] L.A. O'NEILL, Designer's Workbench - Efficient and Economical Design Aids, *Proceeding of 16th Design Automation Conference*, June 1979, 185-199.

[Oes80] D. Bjoerner & O.N. Oest, ed., *Towards a Formal Description of Ada*, Springer-Verlag, 1980.

[Ost81] J. OSTERHOUT, Caesar: An Interactive Editor for VLSI Circuits, *VLSI Design 4*, (November 1981), 34-38.

[PiA86] P. PISTOR and F. ANDERSON, Designing a Generalized NF2 Model with an SQL-Type Language Interface, *Proceedings of the 12th International Conference on Very Large Data Bases*, August 1986.

[Rei84] M. REIMER, Implementation of Database Programming Language Modula/R on the Personal Computer LILITH, *Software Practice and Experience 14*, 10 (October 1984), 945-956.

[RiC87] J.E. RICHARDSON and M.J. CAREY, Programming Constructs for Database System Implementation in EXODUS, *SIGMOD 87 Conference Proceedings*, 1987, 208-219.

[RHM] A. ROSENTHAL, S. HEILER and F. MANOLA, An Example of Knowledge-Based Query Processing in CAD/CAM DBMS, *Proceedings of the 10th International Conference on VLDB*, , 363-370.

[Rub87] S.M. RUBIN, *Computer Aids for VLSI Design*, Addison Wesley , 1987.

[San82] J.L. SANBORN, Evolution of the Engineering Design System Data Base, *Proceedings of the 19th Design Automation Conference*, June 1982.

[Sch80] J.W. SCHMIDT, Programming Languages and Datatabase Models: On the Integration of Concepts, Constructs and Notations, in *Database Systems*, B. Shaw (ed.), University of Newcastle upon Tyne, 1980.

[ScM80] J.W. SCHMIDT and M. MALL, Pascal/R Report, Technical Report No. 66, Fachbereich Informatik, University of Hamburg, 1980.

[Seq83] C.H. SEQUIN, Managing VLSI Complexity: An Outlook, *Proceedings of IEEE 71*, 1 (January 1983), .

[Shi81] D.W. SHIPMAN, The Functional Data Model and the Data Language DAPLEX, *ACM Trans. Database Systems 6*, 1 (March 1981), 140-173.

[Sin83] N. SINGH, MARS: A Multiple Abstraction Rule-based Simulator, Stanford University Heuristic Programming Project HPP-83-43, December 1983.

[SmS77] J.M. SMITH and D.C.P. SMITH, Database Abstractions: Aggregation and Generalization, *ACM Trans. Database Systems 2*, 2 (June 1977), 105-133.

[SFL83] J.M. SMITH, S.A. FOX and T.A. LANDERS, ADAPLEX: Rationale and Reference Manual, T.R. # CCA-83--08, Computer Corporation of America, May 1983.

[Sow81] J. F. SOWA, A Conceptual Schema for Knowledge-Based Systems, *SIGMOD Record 11*, 2 (Feb 1981), .

[SMF86] D.A. SPOONER, M.A. MILICIA and D.B. FAATZ, Modeling Mechanical CAD Data with Data Abstraction and Object-oriented Techniques, *Proceedings of the International Conference on Database Engineering, Los Angeles*, 1986, 416-425.

[Sto74] M. STONEBRAKER, The Design and Implementation of INGRES, *ACM Transactions on Database Systems 1*, 3 (1974), 189-222.

[Str84] B. STROUSTRUP, Data Abstraction in C, Computing Science Technical Report No. 109, AT&T Bell Laboratories, January 1984.

[Suf82] B. SUFFRIN, Formal Specification of a Display-oriented Text Editor, *Science of Computer Programming 1*, 3 (May 1982), .

[Ten76] R.D. TENNENT, The Denotational Semantics of Programming Languages, *Communications of ACM 19*, 8 (1976), 437-453.

[Ter83] C.J. TERMAN, RSIM - A Logic-level Timing Simulator, *Proceedings of IEEE International Conference on Computer Design*, October 1983, 437-440.

[Tic87] W.F. TICHY, What can Software Engineers Learn from Artificial Intelligence?, *Computer*, November 1987, 43-54.

[TsZ86] S. TSUR and C. ZANIOLO, LDL: A Logic-based Data Language, *Proceedings of the 12th International Conference on VLDB*, August 1986, 33-41.

[Weg87] P. WEGNER, Dimensions of Object-Based Language Design, *OOPSLA'87 Proceedings*, 4-8 October 1987, 168-182.

[WeM81] D. WEINREB and D. MOON, Objects, Message Passing and Flavors, in *Lisp Machine Manual*, Symbolics Inc., Cambridge, MA, 1981.

[Wil82] D.S. WILE, Program Developments: Formal Explanations of Implementations, Technical Report No. RR-82-99, USC Information Sciences Institute, August 1982.

[Wir77] N. WIRTH, Program Development by Stepwise Refinement, *Communications of ACM 14*, (April 1977), 221-227.

[Zan83] C. ZANIOLO, The Database Language GEM, *SIGMOD 83/SIGMOD Record 13*, 4 (May 1983), 207-218.

[Zan85] C. ZANIOLO, The Representation and Deductive Retrieval of Complex Objects, *Proceedings of VLDB 85*, 1985, 458-469.

[ZdW86] S.B. ZDONIK and P. WEGNER, Language and Methodology for Object-oriented Database Environments, *Proceedings of the 19th Hawaii International Conference on System Sciences*, 1986, 378-387.

[Zeh86] A. Diener and C.A. Zehnder, ed., *The Federative Database Server*, Institut fur Informatik, ETH, August 1986.

[Zil84] S.N. ZILLES, Types, Algebras and Modelling, in *On Conceptual Modelling*, M.L. Brodie, J. Mylopoulos & J.W. Schmidt (ed.), Springer-Verlag, 1984, 441-449.